06-12-00

A

EXPRESS MAIL NO. EL408130855US

PATENT APPLICATION

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

UTILITY PATENT APPLICATION TRANSMITTAL LETTER

Atty./Agent Docket No.: CS 10246

Mailing Date: Herewith

Express Mail Label No.:EL408130855US

To: Assistant Commissioner for Patents
Box Patent Application
Washington, D.C. 20231

Dear Sir:

Transmitted herewith for filing under 37 CFR 1.53 (b) is a Nonprovisional Utility Patent:

___X_ New Application; or a _Continuation,_____Division, or ____ Continuation-in-Part (CIP) Application of prior US application No._____/_____, filed on _____, having US Examiner _____, in Group Art Unit_____: of

Inventor(s): **Sheila M. Rader, Brian Lucas, Pradeep Garani and Franz Steininger**

For (Title): **Integrated Processor Platform Supporting Wireless Handheld Multi-Media Devices**

This transmittal letter has 2 total pages.

Enclosed are:

__X__ __7 sheets of drawings, along with ___86_pages of specification and claims, Appendix A (94 pages) and Appendix B (11 pages)

__x__ Unsigned Oath or Declaration Combined with Power of Attorney (3 pages)
___ Newly Executed (original or copy)
_____ Copy from a prior application (if this is a Continuation/Division with no new matter)
___ Statement deleting named inventor(s) in prior application if this is a Continuation/Division (See 37 CFR 1.63(d)(2) and 1.33(b).)
___ Consider as the above Statement, Please delete as inventors for this application the following inventors named in the prior application: _____.

_____ A certified copy of a _____ (non-US) application S/N___/_, having a filing date of_____, and foreign priority to this non-US application for the present application is hereby claimed under 35 USC 119.

_____ An Assignment Transmittal Letter and Assignment of the invention to MOTOROLA, INC.

__x__ An Information Disclosure Statement (IDS), with one (1) PTO-1449, and ____1(one)__ citation copies.

_____ Preliminary Amendment

__X__ Return Receipt Postcard

_____ Petition For Extension of Time for parent application of the present Continuation/Division/CIP

Instructions:

_____ Incorporation by Reference (for Continuation/Division application) The entire disclosure of the prior application, from which a copy of the oath or declaration is supplied, is considered as being part of the disclosure of the accompanying application and is hereby incorporated by reference therein.

_____ Since the present application is based on a prior US application, please amend the specification by adding the following sentence before the first sentence of the specification: "The present application is based on prior US application No. _____/_____, filed on_____, which is hereby incorporated by reference, and priority thereto for common subject matter is hereby claimed."

_____ Please cancel filed claims _____.

__X__ The filing fee is calculated as follows:

CLAIMS AS FILED, LESS ANY CANCELED BY AMENDMENT

| | NUMBER OF CLAIMS | NUMBER EXTRA | RATE | FEE |
|---|---|---|---|---|
| TOTAL CLAIMS | 37- 20 = | 17 | X $18 | = $306.00 |
| INDEPENDENT CLAIMS | 8- 3 = | 5 | X $78 | = $390.00 |
| MULTIPLE DEPENDENT CLAIMS | | | $260 | = $ 0.00 |
| | | | BASIC FEE | = $ 690.00 |
| | | | **TOTAL FILING FEE** | **=$ 1,386.00** |

__X__ Please charge Deposit Account No. 13-4768 in the amount of $__1,386.00__ for the Total Filing Fee. Two duplicate copies of this sheet are enclosed.

__X__ The Commissioner is hereby authorized to charge any additional fees which may be required now or in the future under 37 CFR 1.16 or 37 CFR 1.17, including any present or future time extension fees which may be required, or credit any overpayment to Deposit Account No. 13-4768. Two duplicate copies of this sheet are enclosed.

Date: June 9, 2000

Signature: _____

Please forward all correspondence to:
Ray Warren (MCS)
Motorola, Inc.
Personal Communications Sector
600 North US Highway 45
Libertyville, IL 60048

Printed Name: Michael C. Soldner
Attorney for Applicant(s)
Registration No. 41,455
MOTOROLA, INC.
Phone: (847) 523-2585
Fax: (847) 523-2350

Express Mail No. EL408130855US

CS-10246

United States Patent Application

of

Rader, Sheila
Lucas, Brian
Garani, Pradeep
and
Steininger, Franz

# INTEGRATED PROCESSOR PLATFORM SUPPORTING WIRELESS HANDHELD MULTI-MEDIA DEVICES

# INTEGRATED PROCESSOR PLATFORM SUPPORTING WIRELESS HANDHELD MULTI-MEDIA DEVICES

## BACKGROUND OF THE INVENTION

The present invention relates to wireless handheld multi-media devices, such as digital telephones, and more specifically to processor platforms in wireless handheld multi-media devices. Even more specifically, the present invention relates to such processor platforms having minimal size and power consumption and that enable efficient data transfers between multiple processors of the processor platform and multiple peripherals.

New standards for digital cellular systems incorporate high speed packet data network capability in addition to traditional circuit switched voice and data channels. At the same time, among the general public, there is wide spread use of the Internet which offers a host of personal communication, information, electronic commerce and entertainment services. The next generation cellular systems offers the opportunity to market wireless products which have voice, data, and personal information management capabilities, i.e. multi-media devices. These products are destined to become portable information appliances with the potential for significant market share.

In such multi-media devices, in particular digital cellular telephones, processor platforms include two main processor cores: a digital signal processor (DSP) core coupled to the radio interface and a host processor core for running the device and coordinating data movements from several peripherals. Such a device may include as peripherals, a Universal Serial Bus (USB), a Universal Asynchronous Receiver/Transmitter (UART) with an optional mode to support the IrDA standard, a Synchronous Serial Interface (SSI), a Multi-Media Card (MMC), and a Bluetooth interface supporting the Bluetooth standard.

It is desirable to be able to move data to and from the various peripherals and the memory of the host processor, and also to and from the various peripherals and the memory of the DSP, and furthermore, to and from the memory of the DSP and the memory of the host processor. Using a technique known in the art as Direct Memory Access (DMA), such transfers advantageously take place without involving either the host processor or the DSP. Thus, for example, instead of the host processor initiating a data transfer from a particular peripheral to the host processor memory, a DMA controller performs the data transfer, allowing the host processor to focus on more important functions. Advantageously, the DMA technique relieves the host processor and the DSP from the cumbersome tasks of simple data transfers, enabling faster and more efficient use of the processors within the device.

However, a DMA controller forms a hardwired unidirectional data channel between two nodes. The DMA controller is coupled between a particular peripheral and the system bus which accesses both the processor to be relieved of the task of performing the data transfer and it's memory. The DMA controller provides the hardware to implement the direct memory access. Because each data channel is unidirectional, two separate DMA data channels are required for bidirectional data transfers between the two nodes. Furthermore, since each data channel is implemented in hardware, once established, the data channel may not be reconfigured to allow a data transfer to and from different nodes or in a different direction.

Thus, separate unidirectional data channels must be hardwired to allow direct memory access for multiple processors and multiple peripherals. Disadvantageously, in handheld multi-media devices, there may be a large number of peripherals; thus, requiring many DMA controllers to hardwire all of the possible DMA connections. For example, to adequately relieve the host

processor and the DSP from having to perform data transfers between the peripherals and the respective memories, DMA controllers must be implemented in hardware between each peripheral and the host processor memory and the DSP memory, such that each DMA controller establishes the desired unidirectional data channels.

Disadvantageously, in small handheld applications, implementing a large number of DMA controllers expends valuable real estate on the processor platform. In other words, the more hardware DMA controllers needed, the more transistors are required on the processor platform and the more space is consumed on the platform by the DMA hardware. What is needed is a processor platform that implements DMA functionality to allow efficient operation of multiple processors without using traditional DMA hardware for all of the various data transfer paths.

Another concern in processor platforms for small handheld multi-media processors is minimizing power consumption. Employing a processor platform without concern for saving power unnecessarily reduces the battery life, which is important in handheld applications because this decreases the time in between battery charges that are required. Furthermore, in multi-media applications which require a large random access memory (RAM), it is desirable to employ dynamic RAM (DRAM) as opposed to static RAM (SRAM), since DRAM is much less costly than SRAM in terms of die size versus array density.

Additionally, embedded DRAM (eDRAM), which is DRAM embedded on the processor platform, may be used to reduce the overall space required by the processor platform. However, in comparison to SRAM, both DRAM and eDRAM must be periodically refreshed in order to ensure that the data contained therein is saved. The refreshing process, typically performed by a refresh controller, consumes valuable power to make sure that data remains

stored. What is needed is a method to refresh the DRAM in such a way as to conserve as much power as possible.

Furthermore, in such handheld multi-media devices, such as telephones, liquid crystal displays (LCDs), such as those found in personal digital assistants (PDAs), are implemented to allow the user to readily view web pages, for example. A typical LCD requires data to be moved from the video buffer to the display driver circuit. This presents problems in that the large LCD bus must transmit and receive data from 8, 16, or 32 bit busses from a memory (e.g. eDRAM) that is only 8, 16 or 32 bits wide. Disadvantageously, the LCD controller and image processor of the LCD spend much time using the system memory, as a video buffer, relative to other peripherals and devices that are required to access the system memory for DMA techniques, which makes the system memory less accessible to these other peripherals and devices. Thus, when video images are displayed on the LCD, the system memory (e.g. eDRAM) acts primarily as the video buffer and also as the system RAM. A separate RAM (e.g. another eDRAM) may be implemented to act as the video buffer; however, such additional memory disadvantageously adds to the transistor count and thus size of the processor platform. What is needed is an efficient memory that can adequately support an LCD controller and at the same time be used as a system RAM and for DMA data transfers.

The present invention advantageously addresses the above and other needs.

**BRIEF DESCRIPTION OF THE DRAWINGS**

The above and other aspects, features and advantages of the present invention will be more apparent from the following more particular description thereof, presented in conjunction with the following drawings and Appendices A and B wherein:

FIG. 1 is a high level block diagram of a wireless multimedia processor platform having three processor cores: a digital signal processor (DSP), a host processor, and a RISC processor core within an interprocessor communication module (IPCM), wherein the interprocessor communication module performs programmable direct memory access functionality;

FIG. 2 is a detailed block diagram of the wireless multimedia processor platform of FIG. 1;

FIG. 3 is a block diagram of the interprocessor communication module (IPCM) of the wireless multimedia processor platform of FIGS. 1 and 2;

FIG. 4 is a diagram that illustrates the programmably selectable direct memory access (DMA) data channels provided by the interprocessor communication module (IPCM) of FIGS. 1, 2 and 3;

FIG. 5 is an illustration which demonstrates the functionality of the event scheduler of FIG. 2 in accordance with an embodiment of the present invention.

FIG. 6 is a diagram of the pointers and memory buffers within the interprocessor communication module and the host processor memory and which are used for each of the programmable direct memory access data channels;

FIG. 7 is a flowchart of the steps performed in implementing a selective refresh technique performed by a refresh controller of FIG. 2 in accordance with one embodiment of the present invention;

FIG. 8 is a flowchart of the steps performed by the refresh controller of FIG. 2 in performing the temperature compensated method of memory refresh in accordance with another embodiment of the present invention; and

FIG. 9 is a block diagram of a memory refresh system using the selective refresh technique and the temperature compensated refresh techniques of FIGS. 7 and 8.

Corresponding reference characters indicate corresponding components throughout the several views of the drawings.

## DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

The following description of the presently contemplated best mode of practicing the invention is not to be taken in a limiting sense, but is made merely for the purpose of describing the general principles of the invention. The scope of the invention should be determined with reference to the claims.

The present invention advantageously addresses the needs above as well as other needs by providing a wireless multimedia processor platform including multiple processor cores, multiple peripherals, and an interprocessor communication module that performs programmable direct memory access functionality for programmably selectable data transfers between the memories of the multiple processor cores and the various peripherals attached thereto. Furthermore, the processor platform includes features that minimize the size and power consumption of the processor platform as well as allow video buffering from the host processor memory to support an LCD display without limiting the ability of other devices to access the host processor memory.

### Multi-Media Processor Platform Overview

Referring first to FIG. 1, a high level block diagram is shown of a wireless multimedia processor platform having three processor cores: a digital signal processor (DSP), a host processor, and a RISC processor core within an interprocessor communication module (IPCM), wherein the interprocessor communication module performs programmable direct memory access data transfers. Shown is a multimedia processor platform 100 including a digital signal processor 102 (also referred to as DSP 102), a random access memory 104 (also referred

to as RAM, DSP memory or DSP RAM), a host processor 106 (also referred to as host 106 or MCore), an embedded dynamic random access memory 108 (also referred to as eDRAM 108 or host processor memory), a Universal Serial Bus 110 (also referred to as USB 110), a Universal Asynchronous Receiver/Transmitter 112 with an optional mode to support the IrDA standard (also referred to as UART/IrDA 112), a Multi-Media Card 114 (also referred to as MMC 114), and an interprocessor communication module 116 (also referred to as IPCM 116).

The multimedia processor platform 100, which may also be referred to as a processor platform 100 or simply processor 100, is in the form of a single integrated circuit or chip having three processor cores: the DSP 102, the host processor 106, and a RISC processor core within the IPCM 116. The processor 100 represents a "system on a chip" design (also referred to as "SOC"). In the application of a wireless handheld multimedia devices, it is advantageous that the components of the processor 100 all appear on a single chip. This is because of the size constraints in the handheld applications and also due to the fact that separate processors on separate chips must be hardwired together which may result in losses from wiring chip to chip. However, it is important to note that the present invention is not limited to a system on a chip design, and thus, the three processor cores, i.e. the DSP 102, the host processor 106, and RISC processor core of the IPCM 116, may be implemented on separate chips if so desired in another application.

In operation, the IPCM 116 couples all three of the DSP portion (i.e the DSP 102 and RAM 104), the host portion (i.e. the host processor 106 and the eDRAM 108) and the peripherals (i.e. the USB 110, the UART/IrDA 112, and the MMC 114) together. Advantageously, the IPCM 116 provides programmable direct memory access (DMA) data channels to allow direct memory access data transfers (1)

from a particular peripheral to either the DSP RAM 104 or the eDRAM 108, (2) from the DSP RAM 104 or the eDRAM 108 to a particular peripheral, and (3) between the DSP RAM 104 and the eDRAM 108. These DMA data transfers may be time coordinated (occurring at predetermined times) or event coordinated (occurring upon request or instruction). The IPCM 116 has a reduced instruction set computer (RISC) processor core that enables direct memory access (DMA) data transfers over programmably selectable DMA data channels. For example, the IPCM 116 replaces a large number of hardware DMA controllers to create, for example, 32 programmable data channels, wherein each data channel may be programmed to transfer data in any one of the three directions above. An equivalent hardware DMA implementation generally would require 2*n*(c+1) individual DMA data hardware controllers, where the number 2 represents data channels in two directions, n represents the number of data channels and c represents the number of processors capable of using the IPCM 116, such that c+1 is the number of data paths. In the embodiment shown, n=32, c=2 (i.e. the DSP 102 and the host processor 106) such that c+1=3 (i.e. the number of data paths, e.g. peripheral to memory, memory to peripheral and memory to memory), which results in the IPCM 116 replacing of 192 individual hardware DMA controllers. Disadvantageously, this large number of DMA controllers would not be cost effective and would dominate the available space of the processor. Thus, the IPCM 116 provides a number of DMA data channels or data paths that are each configurable or programmable to connect different nodes together. As such, a node is typically the RAM 104, eDRAM 108, or one of the peripherals. Thus, a single DMA data channel may be programmed to provide a direct memory access data transfer from the USB 110 to the eDRAM 108, and then the same data channel may be reprogrammed or reconfigured at

a later time to provide a direct memory access data transfer to from the DSP RAM 104 to the MMC 114.

This is a departure from hardware DMA controllers which provide a non-programmable, unidirectional wired data channel from one node to another node, which may not be later reconfigured as between two different nodes. Advantageously, in a wireless multi-media application, where the physical size of the chip or processor 100 is important and there are a large number of peripheral devices requiring many DMA controllers for direct memory access, the IPCM 116 provides a programmable DMA functionality in which a data channel is formed that may be altered as needed. Thus, many DMA controllers are not needed and the overall size of the processor 100 may be made smaller than if many hardware DMA controllers were implemented.

The function of the IPCM 116 advantageously relieves both the host processor 106 and the DSP 104 from having to stop performing tasks in order to perform data transfers. The IPCM 116 performs these transfers in order to provide the most efficient operation of both the DSP 102 and the host processor 106. In other words, the DSP 102 and the host processor 106 are allowed to operate at their optimal speeds and perform the critical tasks that they were designed for without slowing down to perform simple data transfers.

Referring next to FIG. 2, a detailed block diagram is shown of one embodiment of the wireless multimedia processor platform of FIG. 1. Shown is the multi-media processor platform 100 including the digital signal processor 102 (DSP 102), a DSP RAM 104, DSP peripheral interface 202, the host processor 106 (also referred to as the call processor 106), the embedded DRAM 108 (eDRAM 108), an LCD controller 204 (also referred to more generically as a display controller) including image processing 206 and configuration registers 208, and host peripheral interface 210. The processor platform 100

also includes the following peripherals 230: the USB 110, the UART/IrDA 112, a Synchronous Serial Interface 212 (also referred to as an SSI 212), the MMC 114, and a bluetooth interface 214. Also included is the IPCM 116, an event timer 216, data path select 218 (also referred to as the data path select unit 218), an eDRAM refresh controller 220 (also referred to as refresh controller 220), and a bus interface 222.

Further included are the following busses: a clock bus 224 (also referred to as the "c" bus 224) coupled to the refresh controller 220; the DSP system bus 226 (also referred to as the "d" bus 226) coupling the DSP 102, the DSP RAM 104 and the DSP peripherals 202 to the IPCM 116; the event bus 228 (also referred as the "e" bus 228) coupling the peripherals 230 to the event timer 216 and the IPCM 116; the host system bus 232 (also referred to as the "h" bus 232) coupling the host processor to the peripherals 230 and 210, the LCD controller 204, bus interface 222, data path select 218, refresh controller 220, and the IPCM 116; the IPCM bus 234 (also referred to as the "i" bus 234) coupling the IPCM 116 to peripherals 230; the memory bus 236 (also referred to as the "m" bus 236) coupling the data path select 218 to the eDRAM 108; LCD bus 238 (also referred to as the "p" bus 238) coupling the image processing 206 to the data path select 218; the refresh bus 240 (also referred to as the "r" bus 240) coupling the refresh controller 220 to the data path select 218; an external system bus 242 (also referred to as the "s" bus 242) coupling the bus interface to, for example, external memory 244; and a transfer bus 246 (also referred to as the "t" bus 246) coupling the IPCM 116 to the data path select 218.

Also illustrated in FIG. 2, but not part of the processor platform 100, are the radio hardware 248 coupled to the DSP peripheral interface 202, the LCD panel 250 coupled to the image processing 206, the

external memory 244 coupled to the bus interface 222, and a clock input 252 coupled to the refresh controller 220.

The processor platform 100 includes three processor cores: the DSP 102, the host processor 106, and a RISC processor core embedded within the IPCM 116. Advantageously, in this embodiment, the processor platform 100 is implemented as a system on a chip, although many features of the present invention are not limited to an integrated system on a chip design. The following is a brief description of several of the components of the processor platform 100.

The DSP 102, as known in the art, is interfaced via a DSP system bus 226 to a system memory, e.g. DSP RAM 104, and DSP peripheral interface 202. The DSP RAM 104 contains DSP program and data storage areas. The DSP peripheral interface 202 is used to interface the DSP core (i.e. modem) to the radio hardware 248 to implement cellular radio communications. These components are well known in the art and are commonly found in cellular telephones.

The host processor 106 is a general purpose reduced instruction set computer (RISC) processor or a complex instruction set computer (CISC) processor as known in the art. The LCD controller 204 is a module containing digital logic configured to render an image onto an external LCD panel 250 from a binary bit image contained within memory, e.g. an eDRAM 108 memory array. The host peripheral interface 210 includes one or more modules containing digital logic and configured as a peripheral operated by the host processor 106. Examples of such peripheral interfaces include keyboard interface, general purpose timer, and general purpose I/O ports.

Also included are multimedia peripherals 230. The USB 110 is a logic block configured as a peripheral implementing the media access layer functions of the open standard known as the Universal Serial Bus. The module is configured with 2 interface ports: a port interfaced

to the host processor 106 via the h bus 232 for purposes of configuration management and control and a data port interfaced to the IPCM 116 via the i bus 226 used to pass data to and from the external serial interface.

The IrDA/UART 112 is a logic block configured as a peripheral implementing the necessary functions known in the art as a Universal asynchronous Receiver/Transmitter with an optional mode to support the IrDA standard. The IrDA/UART 112 is configured with 2 interface ports: a port interfaced to the host processor 106 via the h bus 232 for purposes of configuration management and a data port interfaced to the IPCM 116 via the i bus 236 used to pass data to and from the external serial interface.

The SSI 212 is a logic block configured as a peripheral implementing the necessary functions known in the art as a Synchronous Serial Interface. The module is configured with 2 interface ports: a port interfaced to the host processor 106 via the h bus 232 for purposes of configuration management and a data port interfaced to the IPCM 116 via the i bus 236 and used to pass data to and from the external serial interface.

The MMC 114 is a digital interface designed for the purpose of connecting to and operating with a MULTIMEDIACARD. The MMC 114 is configured with 2 interface ports: a port interfaced to the host processor 106 via the h bus 232 for purposes of configuration management and a data port interfaced to the IPCM 116 via the i bus 236 and used to pass data to and from the external MULTIMEDIACARD.

The bluetooth interface 214 is a digital interface containing designed for the purpose of supporting the BLUETOOTH open standard. The bluetooth interface 214 is configured with 2 interface ports: a port interfaced to the host processor 106 via the h bus 232 for purposes of configuration management and a data port interfaced to the IPCM 116 via the i bus 236 and

used to pass data to and from an external Bluetooth compliant radio circuit.

The event timer 216 is a binary counter coupled to the IPCM 116 via the e bus 228 and having 'n' number of states connected to comparison logic capable of detecting a predetermined distinct state for the purpose of generating a signal when a specific state is indicated. The counter is clocked by an accurate clock source. In one embodiment, the event timer 216 can handle up to 32 events at a given time. In some embodiments, the event timer 216 may be used to signal a programmed event or direct memory access data transfer to the IPCM 116, for example, at regular intervals.

The IPCM 116 is designed to perform interprocessor and serial I/O communication employing DSP 102 processor, and the I/O peripherals 230, such as USB 110, IRDA/UART 112, bluetooth interface 214 and MMC 114. The IPCM 116 is strategically positioned in the architecture to enable programmable direct memory access (DMA) data transfers to/from I/O peripherals with either the host processor 106 or the DSP 102. In addition, the IPCM 116 supports data transfers between the DSP and Host itself. Thus, advantageously, the IPCM 116 behaves as a programmable DMA to transfer data into memory without involving either the DSP 102 or the host processor 106 to make the data transfer.

Within the IPCM 116 is a programmable RISC core which is programmed to perform DMA functions. A more detailed description of the inner workings of the IPCM 116 is described with reference to FIG. 3. This approach offers several advantages over non-programmable (e.g. hardware based) solutions. The programmable DMA functionality of the IPCM 116 has many benefits. First, this allows dynamic routing in that the RISC core can be programmed to perform a variety of tasks. Under programmed control, in one embodiment, the IPCM 116 can

configure up to 32 simultaneous DMA data channels, each of which may be configured in any one of three directions. DMA data transfers are routed over respective ones of each of these 32 DMA data channels. For example, each data channel could be configured to as a DMA data channel for DMA data transfers between (1) a peripheral 230 to memory (e.g. DSP RAM 104 or eDRAM 108), (2) memory (e.g. DSP RAM 104 or eDRAM 108) to peripheral 230, or (3) memory (e.g. DSP RAM 104 or eDRAM 108) to memory (e.g. the other of eDRAM 108 or DSP RAM 104). Advantageously, each of these 32 programmed DMA data channels may later be reprogrammed to allow DMA data transfers between two different nodes.

Furthermore, the IPCM 116 allows more functionality in a smaller footprint. This is because the IPCM 116 replaces many separate wired DMA controllers between the various nodes, i.e. the DSP RAM 104, the eDRAM 108, and the various I/O peripherals 230. This represents a significant savings in gates or transistors needed on the die, which for handheld multimedia applications is important in minimizing processor size. Typical gate counts of DMA channels are approximately 3k gates per channel. In this embodiment, the IPCM 116 becomes a space saving advantage when more than 6 hardware DMA channels are required. Advantageously, in this embodiment, the IPCM 116 behaves as 192 (as described above) separate DMA channels within the footprint of about 6 actual hardware DMA channels.

Additionally, the IPCM 116 is flexible and scalable. The concept lends itself to enhancements for future generation products. The flexible data routing capabilities enable additions to the basic architecture such as hardware based accelerators. Enhancements to the RISC core of the IPCM 116 include adding registers and new instructions to permit the concept to meet future needs.

Also, the programmable RISC processor core of the IPCM 116 provides a common application program interface (API) to be defined, since the programmable RISC core uses virtual control registers which are mapped into the host memory (i.e. eDRAM 108). The API can remain intact when enhancements are made to the IPCM 116 in future generations. This feature increases software reusability.

Another feature is that the IPCM 116 is provided with smart power management such that a sleep mode is entered during periods of inactivity. This is important in handheld applications where battery life is an important concern.

Yet another feature of the IPCM is that the DMA data transfers can be transferred to and from memory (e.g. either DSP RAM 104 or eDRAM 108) using little-Endian format or big-Endian format, as known in the art. This enables the IPCM 116 to communicate with different types of processors configured according to either format. For example, a DSP configured for little Endian format or a DSP configured for big-Endian format can interface with the IPCM 116.

Another component of the processor platform 100 is that since the entire system is implemented on a chip, the RAM of the host processor 106 is an on-chip memory array constructed of DRAM type bit cells as known in the art and referred to as an embedded DRAM or eDRAM 108. The array is configured as 65536 words (64k words) of 128 bits each for a total of 67,108,846 bit cells (64M bit). The eDRAM 108 must be refreshed periodically and this function is accomplished by the refresh controller 220. The 128 bit width of the eDRAM 108 is optimized for the LCD controller 204 as will be described further below. Thus, the eDRAM 108 functions as the host processor memory and an on-chip video buffer for the LCD panel 250. Advantageously, the width of the eDRAM reduces the number

of access cycles consumed by the LCD controller 204 and thereby increase the number of access cycles available to the host processor 106 and IPCM 116.

The bus interface 222 is a module containing digital logic configured to function as a memory controller as known in the art. The module supports external memory 244 interfaced to the processor platform 100 via the s bus 242. The external memory 244 includes a number of discrete memory devices such as SRAM, EPROM, FLASH and DRAM. The external memory 244 is directly accessible by the host processor 106 or indirectly accessible by the DSP 102 via the IPCM 116. This is advantageous because this enables the host processor 106 and/or the DSP 102 to upload and run applications that are too large to be stored in the respective memories of these processors. For example, the DSP 102 may upload and run voice recognition programs stored in the external memory 244. Furthermore, the bus interface 222 allows concurrent processing operations between the host processor 106, the IPCM 116 and the DSP 102; thus, implementing a multiple instruction stream, multiple data stream (i.e. MIMD) on a single integrated circuit realizing the multi-media telephone, for example.

Note that since the eDRAM 108 has its own bus, i.e. the m bus 236, instead of being a part of the host system bus or h bus 232, the host processor 106 may access the external memory 244 through the bus interface 222 while at the same time, the IPCM 116 accesses the eDRAM 108.

The data path select 218 contains arbitration logic and a pre-programmed data multiplexer designed for the purpose of interfacing the data path of the eDRAM 108 to one of 4 entities that may request access to the array. The 4 entities which can request access to the eDRAM 108 are the host processor 106 having an access width of 32 bits, the IPCM 116 having an access width of

32 bits, the LCD Controller 204 having an access width of 128 bits, and the refresh controller 220 for performing 16 ms periodic cycle stealing refresh for 4096 rows as known in the art. In the embodiment shown, each requestor has a unique data path bus width, e.g. the p bus 238 has a width of 128 bits while the i bus 246 and the h bus 232 have a 32 bit width. The data path select 218 automatically configures the appropriate data path depending on the device being granted access. As such, the data path select 218 converts the wide array of 128 bits to support popular 32 bit RISC processor cores, e.g. within the IPCM 116. In all cases the memory address presented by the requestor is taken into account in order to reference the correct data elements from the array. The bus width and access modes for each requestor is shown in Table 1 below.

**Table 1: Data path configuration**

| Requestor | Bus width | Read/Write | Access modes |
|---|---|---|---|
| Host Core | x32 | R/W | x8,x16,x32 |
| IPCM | x32 | R/W | x8,x16,x32 |
| LCD Controller | x128 | Read only | x128 |
| Refresh Controller | 0 | Special refresh cycle | Invokes 4096 bit cells |

Furthermore, in one embodiment, the data path select 218 always stores and retrieves data to and from the eDRAM 108 using the big-Endian format. However, since the IPCM is configured to operate in either little-Endian format or big-Endian format, if the IPCM is operating according to the little-Endian format, the data path select 218 converts the data to and from the IPCM according to the little-Endian and to and from the eDRAM into big-Endian format. As such, the IPCM must inform the data path select which format it is configured as.

The refresh controller 220 generates memory

requests to the eDRAM 108 in order to facilitate periodic cycle refresh of the bit cells within the eDRAM array. However, in preferred embodiments of the present invention, the method of refreshing the eDRAM 108 is performed in such a manner as to minimize power consumption in ways not contemplated by known refreshing techniques. Several specific refreshing techniques that are designed to conserve power consumption are described with reference to FIGS. 7-9.

The following is a description of the various bus interfaces. The clock bus 224 ("c" bus) is a clock input line used to sequence and time the refresh controller 220. The DSP system bus 226 ("d" bus) contains a separate address and data path along with control signals to convey read and write operations to the selected device. In addition, a bus request and bus acknowledge signal is also incorporated to allow the IPCM 116 to request use of the DSP system bus. The event bus 228 ("e" bus) is a group of signals driven by the peripherals 230 and sent to the input event detection device of the IPCM 116 (see FIG. 3) for the purpose of activating a data movement operation. The host system bus 232 ("h" bus) contains a separate address and data path along with control signals to convey read and write operations to the selected device. The h bus 232 can operate independently from all other buses. In addition, a bus request and bus acknowledge signal is also incorporated to allow the IPCM to request use of the h bus 232. The IPCM bus 234 ("i" bus) is the IPCM system bus containing a separate address and data path along with control signals for signaling read and write operations to a specified peripheral 230. The memory bus 236 ("m" bus) is a bi-directional bus and is used to interconnect the eDRAM 108 with the data path select 218 module. The memory bus 236 has a bus width of 128 bits. The LCD bus 238 ("p" bus) is a unidirectional bus having a width of 128 bits and is used to move display image

samples to the LCD controller 204 for display on the LCD panel 250. The refresh bus 240 ("r" bus) is a bus that, when asserted, contains the address of the next row to be refreshed in the eDRAM 108 array. The external system bus 242 ("s" bus) contains a separate address and data path along with control signals to convey read and write operations to the selected external memory 244. The external system bus 242 is accessible by the host processor 106 or the IPCM 116. The transfer bus 246 ("t" bus) conveys access requests from the IPCM 116 to the eDRAM 108 array. The transfer bus is bi-directional and has a 32 bit data path and a 32 bit address path.

Another feature of the eDRAM 108 is that it provides an on-chip video buffer as well as being the RAM for the host processor 106. Advantageously, the eDRAM 108 is 128 bits wide, in order to accommodate the LCD controller 204. If the eDRAM were 32 bits wide, which would be customary to support common RISC processor cores and host processors, the eDRAM 108 would be dominated by requests from supporting the LCD controller 204, such that the other devices using the eDRAM would have to compete with the LCD controller 204. Thus, the LCD controller 204 would essentially become the primary user of the eDRAM and the IPCM 116 and the host processor 106 would become secondary user. By providing a very wide buffer that is the same width as the LCD bus 238, the LCD controller 204 only briefly accesses the eDRAM 108 allowing the IPCM 116 and the host processor 106 to become the primary users of the eDRAM 108. Thus, configuring the eDRAM at 128 bits wide, the LCD controller only uses about 2-3% of the eDRAM capabilities, advantageously leaving 97% to the other devices using the eDRAM. Note that the data path select 218 allows for the differently sized busses to access the eDRAM 108.

The fact that the eDRAM 108 doubles as the system memory and the video buffer further reduces space

on the processor platform 100. If a separate dedicated video buffer was employed, such as traditionally done, this separate video buffer would occupy additional space on the processor platform or be a separate integrated circuit wired to the LCD controller 204, which would take up even more space and introduce losses in the wiring. In contrast to conventional discrete video buffers, the eDRAM 108 of the processor platform 100 acts as both the system memory and the video buffer within a small footprint.

The following describes various other features of the IPCM 116.

Since the IPCM allows DMA data transfers from the peripherals mapped to the IPCM bus 236 to the either the DSP RAM 104 or the eDRAM 108, the IPCM allows these peripherals 230 to be used by the DSP 102 and/or the host processor 106. For example, a data storage device, such as the MMC 114 is accessible to the DSP 102 or host processor 106. Thus, MP3 formatted data may be streamed from the MMC 114 to the DSP 102 to affect an Internet audio player. Other applications include using the IPCM to write or read data files located on the MMC 114 by the host processor 106. Furthermore, the IPCM can be configured to move digital audio samples to and from the DSP RAM 104 to a pair of USB isochronous ports; thus, realizing a speakerphone. Additionally, the IPCM 116 supports multiple data streams originating and/or terminating from either the eDRAM 108 or the DSP RAM 104. The data transfers from each of the eDRAM 108 and the DSP RAM 104 may be performed independently of each other.

Additionally, since the IPCM 116 contains a RISC processor core (described in more detail with reference to FIG. 3), it is smart programmable. Thus, program tasks can be off-loaded from the host processor 106 via DMA data transfers from the eDRAM. For example, the IPCM 116 may perform "bit-blit" tasks, normally

performed by the host processor 106. "Bit-blit" tasks, as known in the art of computer graphics, involve altering the background displays of a visual display or causing images to "fly" across a visual display. The IPCM 116 can load the necessary program from the host processor 106 to perform such tasks, instead of the host processor 106 so that the host processor 106 is free to perform other tasks.

Another application would be to off-load the host processor 106 or the DSP 102 from performing packet protocol framer functions such as "PPP" or the "LAP layer" function used in the IrDA standard. Again, advantageously the host processor 106 and the DSP 102 do not have to perform these functions.

In one embodiment, the IPCM 116 supports an external MPEG decoder coupled to either the SSI 212 or the UART/IrDA 112 by sending packets to the decoder under the control and supervision of the host processor 106. The IPCM 116 retrieves the decoded pixel data from the external MPEG decoder and deposits the pixel data into the video buffer area allocated within the eDRAM 108. As such, a "picture-in-desktop-window" is provided to the LCD panel 250.

The IPCM takes advantage of the event timer 216 so that programmable DMA data transfers can be activated according the event timer 216, in addition to being activated by events triggered by the respective peripherals and/or the DSP 102 and/or the host processor 106. Thus, pre-programmed DMA data transfers will automatically occur at predetermined times.

Furthermore, in order to minimize power consumption, the IPCM is designed to enter a low power mode (i.e. sleep mode) when no events are pending. Thus, the IPCM 116 will not unnecessarily drain battery life in between DMA data transfers.

InterProcessor Communication Module (IPCM)

Referring next to FIG. 3, a block diagram is shown of the interprocessor communication module (IPCM) of the wireless multimedia processor platform 100 of FIGS. 1 and 2. Shown is the IPCM 116 (also referred to as a "programmable direct memory access module") including a RISC processor core 302 (also referred to as a RISC core of more generally as processor 302), an event scheduler 304 (also referred to as an event detect unit 304, a task scheduler 304 or a programmable task scheduler 304), static RAM 306 (also referred to as SRAM 306), read only memory 308 (also referred to as ROM 308), DSP direct memory access unit 310 (also referred to as DSP DMA unit 310), a host direct memory access unit 312 (also referred to as a host DMA unit 312), DSP control registers 314 (also referred to as DSP control unit 314), and host control registers 316 (also referred to as host control unit 316). Also shown are the DSP system bus 226 ("d" bus 226), the host system bus 232 ("h" bus 232) the IPCM bus 234 ("i" bus 234), and the peripherals 230 including the USB 110, IrDA/UART 112, SSI 212 and MMC 114. Also shown are the event timer input 318 and peripheral/DMA event inputs 320 into the event scheduler 304 via the event bus 228 ("e" bus 228).

The DSP control registers 314 and the DSP DMA unit 310 are coupled to the d bus 226 via a bus interface. The host control registers 316 and the host DMA unit 312 are coupled to the h bus 232 via a bus interface. The IPCM 116 also includes the i bus 234 which couples to the various peripherals 230. Within the IPCM 116, the RISC processor core 302, the SRAM 306, the ROM 308, the DSP control registers 314, the DSP DMA unit 310, the host control registers 316 and the host DMA unit 312 are all coupled to the i bus 234. Both the DSP DMA unit 310 and the host DMA unit 312 each comprise a bus transceiver portion of a conventional DMA controller. The event scheduler 304 is coupled to the processor 302.

Inputs to the event scheduler 304 are the event timer 318 and the peripheral/DMA events 320.

In operation, the IPCM 116 is provides interprocessor and serial I/O data transfers employing direct memory access (DMA) techniques without actually implementing individually dedicated hardware DMA channels for all the various possible data transfer paths. Advantageously, by providing the IPCM 116 to perform these DMA data transfers, both the host processor and the DSP are relieved of such tasks and can perform more important tasks. Advantageously, and in contrast to traditional DMA circuits (also referred to as DMA controllers) that establish hardwired unidirectional DMA data channels, the IPCM 116 is a programmable DMA module that provides programmable DMA data channels that may be programmed to perform any one of three types of data transfers: (1) from a selectable peripheral 230 to either of two memories (e.g. DSP RAM 104 or eDRAM 108), (2) from either of two memories to a selectable peripheral 230, and (3) between the two memories. Thus, the IPCM 116 configures, for example, 32 programmable DMA data channels, each one which can be configured for one of the six types of data transfers. Advantageously, within the physical footprint of approximately six conventional hardware DMA controllers as known in the art, in one embodiment, the IPCM 116 replaces 192 individual DMA controllers and has the ability to configure 32 out of 192 possible DMA data channel configurations at any given time. Each of these 32 programmed DMA data channels are then used for DMA data transfers. Furthermore, these 32 data channels may then be re-configured to a different 32 out of the 192 possible DMA data channel configurations at a later time or as needed. This proves very valuable and flexible in space conscious applications, such as in handheld devices.

In one embodiment, one of the 32 DMA channels is reserved as a control channel from the host processor

106 to the IPCM 116. Thus, the IPCM 116 can configure 31 DMA data channels out of 186 possible DMA data channel configurations. Advantageously, this control channel allows the host processor to be able to send a control message to the IPCM to reconfigure one or more of a set of 31 configured DMA data channels into another one of the 186 possible DMA data channel configurations. Even if there is no control channel, the entire set of 32 configured DMA data channels may be dumped and reconfigured by the host processor.

In order to accomplish this programmable DMA data transfer capability, the IPCM 116 includes a RISC processor core 302 and also ROM 308 and the SRAM 306. In some embodiments, the RISC processor core 302 comprises a microRISC processor core. The RISC processor core 302 is used to execute short routines or instructions (stored in SRAM 306) which perform DMA data transfers. A specific example, of a custom RISC processor core and its instruction set are further described later in this specification. Also included are a pair of DMA units, DSP DMA unit 310 and host DMA unit 312, interface with the RISC processor core 302 and use specialized, dedicated registers for all DMA transfers. Thus, the DSP DMA unit 310 and the host DMA unit 312 comprise the bus transceiver portion of a conventional DMA controller. The address register, data register and counter, for example, of the conventional DMA controller are implemented within the RISC processor core 302. As such, the respective DSP DMA unit 310 and the host DMA unit 312 each represent two wired data paths to and from the RISC processor core 302 and the respective busses, e.g. d bus 226 and h bus 232.

The ROM 308 contains startup scripts (i.e. boot code) and the other common utilities which are referenced by scripts that reside in the SRAM 306. An example set of ROM scripts are attached in Appendix B. The SRAM 306 is divided into a processor context area and a code space

area used to store channel scripts. Channel scripts are downloaded into SRAM 306 from the eDRAM or from external memory by the IPCM 116 using the host DMA unit 312. Downloads are invoked using command and pointers provided by the host processor. Each programmable or "virtual" DMA data channel can be configured independently on an "as needed" basis under the control of the host processor. This permits a wide range of IPCM functionality while using the lowest internal memory footprint possible. Microcode routines can be stored in an external memory, e.g. a large capacity Flash memory, and downloaded when needed.

The task scheduler 304 is a programmable scheduler that receives requests from the peripherals 230, host processor 106, and DSP RAM 102 for DMA data transfers. These requests are in the form of "events" detected on the e bus 228. An event is a condition that arises that controls the operation of a particular programmable DMA data channel. For example, an event is an indication from one of the peripherals, the host or the DSP (e.g. peripheral/DMA event inputs 320) that a DMA data transfer is desired. An event may be a signal from the host processor alerting the IPCM to re-program a specific DMA data channel. An event may also be a timed indication from the event timer (i.e. event timer inputs 318) that a DMA data transfer is to take place. For example, depending on which line of the e bus 228 an event is detected on, the task scheduler 304 can tell who is making the request or indicating that a DMA transfer is desired. The task scheduler 304 prioritizes and manages the requests. The task scheduler 304 monitors and detects external events for DMA data transfers, and maps the event (e.g. signal indicating a DMA data transfer is to be performed) to a particular DMA data channel. The events are mapped as DMA data transfers within a specific DMA data channel according to a priority such that higher priority data transfers will

occur before lower priority DMA data transfers. Furthermore, the task scheduler 304 is capable of performing "priority-based preemption" in which a particular DMA data transfer currently being executed by the IPCM is interrupted (i.e. paused) so that a higher priority DMA data transfer may be executed. Once the higher priority DMA data transfer has been completed, the DMA data transfer having been interrupted is then resumed, unless another higher priority DMA data transfer is requested. Priority-based preemption is known to processors generally; however, conventional DMA controllers are hardware-based (i.e. non-programmable) and thus, not capable of such preemption. Advantageously, this embodiment provides priority-based preemption in a programmable DMA system.

The following is a brief description of the data flow in the different types of programmable DMA data transfers supported by the IPCM.

1. Peripheral to Memory

In operation, the various peripherals 230 are responsible for gathering data to be input into the processor platform. When data has arrived at the particular peripheral, for example, at the MMC 114, the peripheral signals an event to the task scheduler 304 of the IPCM 116 via the event bus 228. The task scheduler 304 is able to handle 32 events at any given time. The event is prioritized by the task scheduler 304 and mapped to a particular DMA data channel. Once the event is to be executed, the RISC processor core 302 runs software in the form of scripts located in the SRAM 306. The software is specific to the particular DMA data channel and configures the particular DMA data channel. The software effectively disciplines the RISC processor core 302 to affect the DMA data transfer from the specific peripheral to the memory destination, e.g. either the eDRAM or the DSP RAM. The DMA data transfer is performed

by the software in the RISC processor core 302 such that the data in the peripheral travels to the respective memory via the i bus 234 and the respective DMA unit, e.g. either the DSP DMA unit 310 or the host DMA unit 312.

Advantageously, the DMA data transfer occurs without involvement of the either the DSP or the host processor. Furthermore, by using the IPCM 116 which includes the RISC processor core 302 and a single hardware DMA circuit, e.g. host DMA unit 312, many different DMA data paths are established through a single hardwired DMA unit. Each of these data paths are referred to as a programmable DMA data channel or a "virtual" DMA data channel. For example, there may be a DMA data channel or path from the USB 110 to the DSP RAM 104 and another DMA data channel or path from the SSI 212 to the DSP RAM 104, both of which travel through the DSP DMA unit 310. Advantageously, either DMA data channel may be later reconfigured as a different DMA data channel, e.g. from the MMC 114 to the DSP RAM 104. Thus, each peripheral to memory DMA data channel utilizes either the DSP DMA unit 310 or the host DMA unit, but may be may be programmably selectable as from any one of the peripherals coupled to the IPCM 116.

2. Memory to Peripheral

This type of DMA transfer is opposite the first type in that the transfer is from the memory of one of the processor cores of the processor platform, e.g. the DSP RAM or the host processor memory (e.g. eDRAM) to one of the peripherals 230. The DSP, via the DSP control registers 314, signals an "event" (data transfer) to the task scheduler 304, which prioritizes the event and maps it to a DMA data channel and signals to the RISC processor core 302 to perform the data transfer. The information provided by the DSP indicates a location in the DSP RAM that the data is stored and how much data to

transfer. Then, the RISC processor core 302 runs software in the form of scripts located in the SRAM 306. The software is specific to the particular DMA data channel. The software effectively disciplines the RISC processor core 302 to affect the DMA data transfer from the DSP RAM 104 to the particular peripheral 230. The transfer is performed by the software in the RISC processor core 302 such that the data is copied from the DSP RAM into registers within the DSP DMA unit, then transferred to the peripheral via the i bus 234.

3. Memory to Memory

A third type of DMA data transfer is memory to memory. For example, in the event data is to be transferred from the DSP memory (e.g. DSP RAM 104) to the host processor memory (e.g. eDRAM 108), the DSP would assert an event to the task scheduler 304 of the IPCM 116. The task scheduler 304 recognizes the event, prioritizes it and then causes the RISC processor core 302 to load scripts from the SRAM 306 to affect a DMA data transfer from the DSP RAM to the RISC processor core 302 itself via the DSP DMA unit 310. For example, the data is temporarily placed into registers within the RISC processor core 302. Then, a DMA data transfer is performed between the RISC processor core 302 and the host processor memory (e.g. eDRAM 108) via the host DMA unit 312. This is effectively a "back to back" DMA data transfer. The IPCM 116 resolves differences in a memory sizes. For example, if the DSP RAM is 16 bits wide and the host processor memory is 32 bits wide, the IPCM will gather 16 bit words and pack them into 32-bit words, then transfer the 32-bit words to the host processor memory.

Once the complete "back to back" DMA data transfer has taken place from the DSP RAM to the host processor memory via the RISC processor core 302, the IPCM 116 will signal to the host processor to inform it that there is data stored in its memory. In other words,

the RISC processor core 302 sends a control signal via the host control registers 316 to the host processor, giving the host processor a location pointer to an address in the host processor memory where the data begins and how many words have been placed in the host processor memory starting at that address. At that point, the host processor will retrieve the data at it's convenience. Note that most DMA data transfers are many bytes in length (e.g. 1000 bytes), requiring many iterations before a transfer complete event is signaled. This notification process is also the same in a peripheral to memory transfer, i.e. the RISC processor core 302 notifies the respective processor core, e.g. DSP or host processor, that data is waiting in memory.

This is in contrast to a processor bridge, as known in the art that allows data transfers between two processors. For example, if a host processor wanted to move data from the host to the DSP, the host would have to interrupt the DSP, wait until the DSP was ready to exchange data, then for a brief moment, the host processor would control the DSP memory in order to effect the transfer. This disadvantageously temporarily halts both the DSP and the host processor during the data transfer. Thus, the host memory and the DSP memory each stop and communicate at the same moment.

In contrast, the IPCM 116 allows a direct memory access data transfer from the host memory into the RISC processor core 302 without interrupting the DSP. The only activity required of the host processor 106 is to transmit the control signals to signal an event to the IPCM to perform the DMA data transfer of data from the host memory into the RISC processor core 302. Next, a DMA data transfer is performed from the RISC processor core 302 into the DSP memory. The DSP then retrieves the data from the DSP RAM. In this situation, neither the DSP or the host processor have to stop for the other to cause the transfer.

Referring next to FIG. 4, a diagram is shown that illustrates the programmably selectable direct memory access (DMA) data channels provided by the IPCM of FIGS. 1, 2 and 3. Shown are the IPCM 116, the DSP DMA unit 310, the host DMA unit 312, the RISC processor core 302, the i bus 234, the d bus 226, and the h bus 232. The DSP DMA unit 310 includes a first DSP DMA data connection 402 and a second DSP DMA data connection 404. The host DMA unit 312 includes a first host DMA data connection 406 and a second host DMA data connection 418.

The IPCM 116 includes the DSP DMA unit 310 and the host DMA unit 312. Each DMA unit 310 and 312 comprises a bus transceiver portion of a conventional DMA controller and forms 2 hardwired DMA data connections (through which programmable DMA data channels are established for DMA data transfers), one in the direction of RISC processor core 302 to memory and the other in the direction of memory to RISC processor core 302. These four DMA data connections are programmed by the RISC processor core 302 of the IPCM to act as if they together, with the RISC processor core 302, were 192 (186 if one of the DMA data channels is a control channel) actual hardware DMA controllers. In contrast, conventional DMA controllers only allow one dedicated DMA channel to be established using a DMA data connection.

In a broad sense, the RISC processor core 302 acts as a switch between devices and the various wired DMA data connections. Thus, the first host DMA channel data connection 406 may be configured or programmed as many different programmable DMA data channels, e.g., a DMA data channel from the USB 110 to the eDRAM 108, a DMA data channel from SSI 212 to eDRAM 108, and a DMA data channel from MMC 114 to eDRAM 108. These different DMA data channels utilizing the first host DMA data connection 406 may be referred to as "virtual" DMA data channels, since they effectively provide more DMA data channels than exist in hardware. Thus, the first host

DMA channel 406 is programmably selectable such that it can support DMA data transfers from any one of several peripherals or from the originating node of the second DSP DMA data connection 404 to a memory at the destination end of the first host DMA data connection 406, e.g. the eDRAM 108. Thus, the RISC processor core 302 and a single DMA unit, e.g. host DMA unit 312, replace many separately wired conventional DMA controllers.

Likewise, the second host DMA data connection 408 may be programmably selectable into "virtual" DMA data channels from the originating end or node (e.g. eDRAM 108) and to any one of several peripherals or to the destination node of the first DSP DMA data connection 402. These virtual DMA data channels each utilize the second host DMA data connection 408. Furthermore, a "back to back" DMA data channel may be affected through the second host DMA data connection 408 and the first DSP DMA data connection 402 via the RISC processor core 302.

Custom RISC Processor Core/IPCM

It is noted that the RISC processor core may be a standard RISC processor as is known in the art. However, custom RISC processors may be designed which may improve performance in the IPCM 116. The following is a description of a specific embodiment of a custom RISC processor core and IPCM for use as the IPCM of FIGS. 1-4.

The custom RISC processor core 302 is a 32-bit register architecture with 16-bit instructions. There are 8 general purpose 32-bit registers, 4 flags (T, LM, SF, and DF) and PCU registers (PC, RPC, SPC, and EPC) as known in the art. The RISC processor core 302 is a two stage pipeline and also includes ROM 308 and the SRAM 306. The ROM 308 is 1k byte (configured as 256x32) and the SRAM 306 is 8k byte (configured as 2048x32).

The custom RISC processor core 302 (hereinafter simply referred to as the RISC processor core 302) is

used to execute short routines which perform DMA data transfers. The instruction set (stored in SRAM 306) is comprised of single cycle instructions with the exception of Load/Store, CRC, DMA, and branch instructions which take two, or more cycles, to execute. A preferred instruction set is provided in Appendix A, which is attached hereto. The i bus 234 supports a 32-bit data path and a 16-bit address bus. A pair of DMA units, DSP DMA unit 310 and host DMA unit 312, interface with the RISC processor core 302 and use specialized, dedicated registers for all DMA transfers.

The ROM 308 contains startup scripts (i.e. boot code) and the other common utilites which are referenced by scripts that reside in the SRAM 306. The SRAM 306 is divided into a processor context area and a code space area used to store channel scripts. Channel scripts are downloaded into SRAM 306 from the eDRAM or from external memory by the IPCM 116 using the host DMA unit 312. Downloads are invoked using command and pointers provided by the host processor. Each programmable or "virtual" DMA data channel can be configured independently on an "as needed" basis under the control of the host processor. This permits a wide range of IPCM functionality while using the lowest internal memory footprint possible. Microcode routines can be stored in an external memory, e.g. a large capacity Flash memory, and downloaded when needed.

The task scheduler 304 is responsible for monitoring and detecting external events, mapping events to DMA data channels (also referred to simply as channels) and mapping individual channels to a pre-configured priority. At any point in time, the task scheduler will present the highest priority channel requiring service to the IPCM 116. A special IPCM core instruction is used to "conditionally yield" the current channel being executed to an eligible channel that requires service. If, and only if an eligible channel is

pending will the current execution of a channel be pre-empted. There are two "yield" instructions that differently determine the eligible channels: in the first version, eligible channels are pending channels with a strictly higher priority than the current channel priority; in the second version ("yieldge"), eligible channels are pending channels with a priority that is greater or equal to the current channel priority. The task scheduler 304 detects devices (e.g., channels) needing service through the 32 input event port (the event timer input 318 and the peripheral/DMA events 320). After an event is detected, and only if it is mapped to a channel, the channel event is latched into the "Channel Pending (EP)" register. The priorities of all pending channels are combined with control bits set by the host processor and the DSP and continuously evaluated in order to update the highest pending priority. Each bit in the channel pending register is cleared by the channel script software when the channel service routine has completed.

The Host Control module (i.e. host control registers 316) contains several small RAM blocks organized as an array which are used to control (i.e., channel mapping) the 32 individual channels. The Channel Enable Register is the largest RAM array (32bits X 32bits) and is used to map events to a specific channel(s). The second array is the Priority RAM and is used to assign channels to a programmable 1-of-7 level priority.

The 32 event inputs connected to the task scheduler via the e bus 228 come from a variety of sources and are analogous to interrupt request signals. The receive register full and transmit register empty events that are found in UART and USB ports are typical examples of signals connected to the Event Port on the IPCM. Some of the event inputs are sourced from the Layer 1 timer (e.g. event timer 304). Within the Layer 1 timer are register based compare/capture blocks which can

be used to signal an event for a unique, momentary state of the Layer 1 timer. These events can be used to trigger a specific IPCM channel or channels. This feature can be used to realize a "just-in-time" data exchange between the two processors (e.g. DSP and host processor) to relax the requirement to meet critical deadlines.

The embedded nature of the IPCM requires on-chip debug capability to assure product quality and reliability and to realize the full performance capabilities of the core. The OnCE compatible debug port includes support for setting breakpoints, single step & trace and register dump capability. In addition, all memory locations are accessible from the debug port.

The IPCM 116 has two memory spaces: one for the instructions and one for the data; as both spaces share the same resources (ROM and RAM devices), the system bus manages possible conflicts when the IPCM accesses the same resource for both instruction read and data read or write.

Instructions, that are 16-bit wide, are stored in 32-bit wide devices and are also accessible as data. The correspondence is Big Endian: an even instruction address (terminated by `0') accesses the Most Significant part of the 32-bit data (bits [31:16]) and an odd instruction address (terminated by `I') accesses the Least Significant part of the 32-bit data (bits [15:0]).

Instructions can be fetched from the IPCM ROM and RAM. The ROM, RAM, peripherals (USB, UART1, UART3, MMC and VSAP) and memory mapped registers are accessible as data.

The task scheduler 304 is a hardware based design used to coordinate the timely execution of 32 programmable selectable DMA data channels (virtual DMA channels) by the IPCM on the basis of channel status and priority. The task scheduler performs the following

functions: (1) monitors, detects, and registers the occurrence of any one of the 32 event inputs provided; (2) links a specific event input to a specific channel or group of channels (channel mapping); (3) ignores events which are not mapped to a previously configured channel(s); (4) maintains a list of all channels requesting service; (5) assigns a pre-programmed priority level (1 of 7) to each channel requesting service; and (6) detects and flags overrun/underrun conditions.

A programmable DMA data channel or virtual DMA data channel (hereafter simply called a channel) manages a flow of data through the IPCM 116. Flows are typically unidirectional, but are reconfigurable or reprogrammable. The IPCM can have 32 simultaneously operating channels, numbered 0 to 31. Channel 0 is dedicated for use by the host processor 106 to control the IPCM 116. All other channels can be assigned by the host processor software.

An event is a condition that arises which can control the operation of a channel. Events may be caused by externally (i.e., external to the IPCM) controlled conditions (e.g., UART receive FIFO reaches a threshold) or by the firing of internal timers (e.g. the event timer). The IPCM will implement at most 32 events, which occur randomly with respect to each other. Thus, events are designed to arrive at the task scheduler randomly while the task scheduler can handle 32 events at any one time.

The task scheduler 304 maps events to channels and prioritizes events. A channel can stall waiting on a single event. A single event can awaken more than one channel (e.g., the L1 timer). The mapping from an event to the channels it affects is under program control. There is a register for each of the 32 events which contains a bit map. There is 1 bit for each channel, which determines which channels are awakened by the event. There is also a register for each of the 32

channels which contains the priority at which the channel will operate.

A hardware scheduling block implements a scheduling algorithm such that, when a script executes an instruction that allows rescheduling, the highest priority script that has a pending event will be run.

Multiple channels may be runnable at any given time. The task scheduler 304 (hardwired logic) picks the highest priority channel to run when the current channel yields. Yielding channels may block on an external event or awaiting intervention by the host processor 106 or signal processor 102. The I-th channel is runnable only if the following is true;

(HE[i] ] | HO[i[) & DDE[i] | DO[i]) & (EP[i] | EO[i])

The host enable bit HE[i], for each channel may be set or cleared by the host processor. It can be cleared by a script.

The host override enable bit, HO[i], for each channel may be set or cleared by the host processor. By setting this bit, the host processor 106 may allow channels that do not involve it, like a communication between the DSP 102 and a peripheral 230.

The dsp enable bit, DE[i], which is set or cleared by the dsp. It can be cleared by a script.

The dsp override bit, DO[i], which is set or cleared by the host processor. By setting this bit, the host processor can prevent the DSP from stalling a channel. This will be the case when a channel transfer does not involve the DSP.

The event pending bit, EP[i], which is an output of the task scheduler. It can be cleared by a script. It also can be set by the host to override the event/channel connection matrix.

The event override bit, EO[i], which is set or

cleared by the host processor. By setting this bit, the host processor may prevent a channel from stopping to await peripheral events. This will be the case when the channel is not handling i/o events, e.g., a host processor to DSP DMA data transfer.

All of the HE[i], HO[i], DE[i], DO[i], ER[i], and EO[i] are set to zero on reset.

The IPCM 116 can clear the HE[i], DE[i], and EP[i] bits by means of the done instruction or the notify instruction. The done instruction causes a reschedule while the notify instruction does not. The done and notify instructions can clear one (and only one) of the following bits:

HE[I], DE[I], or EP[I]

When several channels with the same priority are eligible; the hardwired selection tree will automatically select the channel with the highest number: i.e., if channel 7 and channel 24 with priority 4 are both pending, channel 24 will be next channel to run.

In the case of the "yieldge" instruction (i.e. yield if greater or equal), and channels with the same priority as the current channel are pending, the behavior is driven by the hardwired selection tree as described above. For example, given three channels (i.e. 7, 23 and 29) that have the same highest priority.

Channel 7 is active and runs a "yieldge"; it is preempted by channel 29; after a while channel 29 runs a "yieldge", it is then preempted by channel 23 that is the selected channel as channel 29 does not belong to the selectable channels because it is the current channel. Later on, channel 23 runs a "yieldge" and is preempted by channel 29. Channels 23 and 29 will go on switching after every "yieldge" until one of them terminates. It is only at that point that channel 7 becomes eligible. During that example, it is supposed that no other

eligible channel is pending.

Referring next to FIG. 5, an illustration is shown which demonstrates the functionality of a specific embodiment of the task scheduler used in a custom RISC processor of FIG. 3 in accordance with an embodiment of the present invention. Shown is edge detection and latch unit 502, multiplexer 504, counter 506, Channel Enable RAM 508, Channel Pending Register 510 (EP which produces the event pending bit EP[i]), "OR" gate 512, "AND" gate 514, Channel Error Register 516, host enable register 518 (HE which produces the host enable bit HE[i]), host override enable register 520 (HO which produces the host override enable bit HO[i]), dsp enable register 522 (DE which produces the dsp enable bit DE[i]), dsp override register 524 (DO which produces the host override bit DO[i]), event override register 526 (EO which produces the event override bit EO[i]), decision tree 528, priority register 534, highest pending priority register 530 (HPPR), and highest pending current channel register 532 (HPCR).

The task scheduler 304 contains a 3 stage pipeline for processing and prioritizing event inputs. The first stage of the pipeline scans the event inputs and maps detected events to an active channel(s). The second stage of the pipeline maintains a list of channels requesting service (Channel Pending Register) and assigns a priority to all pending channels from the Priority RAM 534 while the third stage identifies the top priority and the associated channel.

The priority output of the task scheduler is applied to the RISC processor core of the IPCM and compared to the priority currently being executed by the RISC processor core. The core maintains the current priority in a Program Status Word (PSW). Priority-based preemption will occur if the task scheduler priority is greater than the current priority when a yield

instruction is encountered.

The following text contains a description of the pipeline of the task scheduler.

The first stage (stage 1) of the pipeline contains a 32 bit edge detection and latch unit 502 placed in front of a 32X1 digital multiplexer 504 (referred to as mux or M1). The mod 32 counter 506 drives the mux 504 select input and the mux 504 sequences through 32 event inputs. A logic "1" is applied to the 32x32 entry Channel Enable RAM 508 for each corresponding logic "1" being scanned; thus mapping an event to a channel. The edge-detect latch corresponding to the event scanned is automatically cleared on the next clock cycle. A positive detect on one of the event inputs supplies a read enable signal to the Channel Enable RAM 508. An event can take up to 32 clocks to be detected. Once detected, an event can propagate from the mux 504 through the Channel enable RAM 508 in 1 clock cycle.

In the second stage, The Channel Enable RAM 508 is addressed by the same 32 bit counter 506 and is a square (32X32) array having a single bit provided for each event across each channel. A logic "1" output is produced from the channel enable RAM 508 on the data output port if a channel is enabled and the corresponding event is detected. This signal is latched into the corresponding position within the Channel Pending Register 510. The detected event propagates from the Channel Enable RAM 508 to the Channel Pending Register 510 in 2 clock cycles. The Channel Pending Register 510 (EP) is a 32 bit register having one bit for each of the 32 channels. Note that the number of channels asserted to the Channel Pending Register 510 on any given clock cycle is determined by the number of bits set to "1" in the 32 bit mask corresponding to the address selected by the 5 bit counter. An "OR" gate 512 is used to provide "present state feedback" to trap and hold a detected

event in the Channel Pending Register 510. Also note that a "bit clear" input is provided to allow the IPCM core to clear a specific bit in the Channel Pending Register 510. This operation is performed by the IPCM core through a software instruction when the corresponding channel has been serviced.

Lastly, a group of 32 "AND" gates 514 is used to detect the reassertion of an event which has not been serviced. In other words, if an event is signaled and the channel is already pending, then a corresponding bit signifying and overrun/underrun condition is latched into a Channel Error Register 516. The host processor is then signaled of any errors in the task scheduler.

In the third stage of the pipeline, the output of former Channel Pending Register 510, as well as output of EO register 526, HE register 518, HO register 520, DE register 522 and DO register 524, and the priority register 534 (which stores the current priorities) to determine which channels are enabled and have to be ordered through the decision tree 528.

This priority tree 528 will deliver, on a cycle by cycle basis, the enabled channel with the highest priority (a channel "i" is said enabled if )EP[i] | EO[i]) & (HE[i] | HO[i]) & (DE[i] | DO[i]) is set to one for this particular channel).

In case multiple enabled channels with the same priority are eligible, the first one detected will be elected (the detection starting with channel 31 and going downwards; i.e., if channel 13 and channel 24 are both eligible, channel 24 is selected).

The "tree architecture" allows dynamic modification of (EP[i] | EO[i]) & (HE[i] | HO[i]) & (DE[i] | DO[i], as well as dynamic modifications of priorities.

The propagation time is one clock from the Channel Pending Register 510 to the FIFO.

The following is a brief summary of the steps of the task scheduler of FIG. 5, referring to numbers corresponding in FIG.5 to the portion of FIG. that performs the step.

At step 1, the Digital mux 502 scans event inputs. All inputs are edged sensed and latched until scanned. After scanning they are automatically cleared. At step 2, a bit in the Channel Pending Register 510 is "set" when the corresponding bit in the event and Channel enable register 508 map produces a positive result. At step 3, the Channel Pending Register 510 (EP) is continuously combined with bits from the EO, DE, DO, HE and HO registers, and together with the priorities, feeds the decision tree 528. At step 4, the value in the highest pending priority register 530 (HPPR) presents the highest priority currently pending service to the RISC processor core. At step 5, the Channel Error Register 516 captures occasions when a channel is pending and a another subsequent event is detected on that channel. At step 6, the IPCM core clears single bit in Channel Pending Register 510 when the channel has been serviced. Once cleared, the channel may be reasserted when another event has been detected.

On execution of a done instruction the program control unit will jump to a context switch subroutine (see Appendix B- EXAMPLE ROM SCRIPTS FOR RISC PROCESSOR CORE OF IPCM, and spill current context to memory. The "real context switch", where CCR (Current Channel Register) (not shown in FIG. 5, but located within the RISC processor core of the IPCM) changes to HPCR 532 (Highest Pending Channel Register), takes place on execution of TstPendingAndSwitch instruction, after the current context has been spilled to memory.

The context switch program (see Appendix B) can be divided into 6 parts: (1) Load current context's spin base address; (2) Spill registers of current context to

memory; (3) Test Pending Channel and react appropriately (TstpendingALnd Switch); (4) Load new context's base address; (5) Download new context; and (6) Copy shadow registers.

On execution of TstPPendingAndSwitch instruction, if there is no longer an HPPR 530 (Highest Pending Priority Register) is null, due to modifications of (HE[i] | HO[i]) & (EP[i] | EO[i] & DE[i] | DO[i] or priorities modifications), the RISC core will enter its IDLE mode, else the second half of the context switch script will be executed, and the context of the elected channel will be downloaded.

In case we enter IDLE mode (i.e., in case HPCR/HPPR is empty after the context spill to memory has finished), on a new event we will continue the second half of the script and download the context of channel associated to the new event.

Once elected, Current Channel Register's status (CCR) and Current Channel Priority (CPRi) can not be modified, even if the associated (HE[i] | HO[i]) & (EP[i] | EO[i] & DE[i] | DO[i] condition transitions to zero or if the priority associated to this channel is dynamically modified by the host.

The yield (done_000) will be handled on a different way in case after the spill part of subroutine, HPCR/HPPR is empty. In this particular case, we will continue the script and download back the context we just spilled to memory.

The following portion describes several of the functional units of the IPCM 116.

In one embodiment, the custom IPCM includes a cyclic redundancy check (CRC) unit (not shown) which is coupled to the RISC processor. It can perform CRC calculation for a set of given polynomials from degree 8 to 32. The CRC unit includes two 32 bit registers: the

CRC algorithm CA which selects the polynomial and the CRC checksum CS to accumulate the data after each processing.

After loading both registers to select the polynomial and initialize the calculation, the CRC unit is able to process one byte every clock cycle.

The DSP DMA unit 310 receives instructions from the RISC processor core 302. It is able to read and write data from/to the DSP memory via DMA accesses and it allows 32-bit and 16-bit transfers to the Quartz DMA.

There are two major registers used by the DSP DMA 310: the address register (DA) that contains the address of the data to read or write in the Quartz memory and the data register (DD) that contains the data itself.

Due to its prefetch and flush capabilities, the DSP DMA 310 is able to perform accesses to the Quartz memory without stalling the RISC processor core so that it can execute other instructions while the DSP DMA 310 waits for the read or write command to complete.

The host DMA unit 312 receives instructions from the RISC processor core 302 (also referred to as the IPCM core). It is able to read and write data from/to the embedded DRAM 108 or the external memory 244 via DMA accesses and it allows 32-bit, 16-bit and 8-bit transfers. The external memory 244 accesses are done through an MCORE local bus (MLB) switch.

There are two major registers used by the Host DMA 312: the address register (MA) that contains the address of the data to read or write in the Quartz memory and the data register (MD) that contains the data itself.

Due to its prefetch and flush capabilities, the Host DMA 312 is able to perform accesses to the host processor (e.g. MCORE) memory (i.e. eDRAM 108) without stalling the RISC processor core 302 so that it can execute other instructions while the Host DMA unit 312 waits for the read or write command to complete. Note

that MCORE is a known host or call processor.

An IPCM bus Switch allows the host DMA unit 312 to access external memories through the MCORE's EIM module connected to the MCORE Local Bus (MLB). It also allows other devices (such as the MCORE Test module, the NEXUS module or the GEM module) to take control of the MLB.

Furthermore, The IPCM 116 internally manages two power modes: RUN and IDLE. In RUN mode, all the modules of the IPCM 116 receive a 100 MHZ IPCM clock. The IPCM 116 is in RUN mode when a channel is active.

In IDLE mode, all the channels have been serviced and no more channels are pending; this is the mode after reset. Most of the IPCM modules do not receive the 100 MHZ clock any more: the RISC processor core, the functional units and the bus arbitration logic. The task scheduler 304 continues to work as it has to sample incoming events and decide when the RISC processor core 302 should start again to service a new channel. The host control modules 316 and DSP control modules 314 that hold the control registers of the IPCM 116 still receive the 100 MHZ clock as they may be accessed by the host processor 106 or the DSP 102 while the IPCM 116 is in IDLE mode. The IDLE mode conserves power, which is important in battery operated environments.

It is also possible for the host processor to completely stop the IPCM 116 through a clock controller. In that case, no clock at all is received by the IPCM 116 and it is not possible for the IPCM 116 to restart its clock. It is up to the host processor to restore the IPCM clock. This functional mode can be useful when no data transfers are required through the IPCM 116.

After reset (either received from the reset module or a software reset required by the host processor), the IPCM 116 is in idle mode; it will start its boot code located at address 0 once a channel is

activated. Activating a channel can be done by the host processor after programming a positive priority and setting the channel bit in the EVTPEND register.

It is important to notice that when the IPCM 116 is in IDLE mode, it is impossible to perform the OnCE debug actions that need to execute instructions on the core; beforehand, the IPCM must be wakened by activating a channel. One possibility is to perform a debug request through the OnCE, and then activate any channel, which will start the IPCM clock and the core will immediately enter debug mode without executing any instruction. More details can be found in the OnCE specific chapter. Another possibility is to use a JTAG module to switch off IPCM clock gating, which enables it to immediately respond and does not require to have an active channel; in that mode, clocks are always running, whatever the IPCM mode.

The following text describes the software interface used to operate the programmable DMA data channels of the IPCM 116.

Referring next to FIG. 6, a diagram is shown of the pointers and memory buffers within the IPCM and the host processor memory and which are used for each of the programmable direct memory access data channels. Shown is the IPCM including table 602 including a pointer register 604, a channel enable register 606, DSP override register 608 and event override register 610. Also shown is the system RAM 601 including Table 612 which includes channel pointers 613. The system RAM 601 also includes Table 614 including a buffer pointer 616, count field 619 and mode field 620. And the system RAM also includes buffers 618.

This section describes buffer management mechanism used between IPCM 116 and either the host processor 106 (also referred to as Host MCU) or DSP 102. The IPCM relies on having a minimum of one or more memory

buffers for each channel that is active. These channel buffers must be located in the system RAM 601 (e.g. eDRAM 108) of the MCU and DSP (e.g. DSP RAM 104) respectively. Buffers are described and managed through the use of virtual control registers implemented as a Linked-List residing in the system RAM 601 area of the MCU and DSP. The following describes the control structure for the Host MCU port.

Table 602 is a group of hardware registers located within the IPCM 116. The pointer register 604 is a 32-bit read/write register accessible by the Host Core that contains a pointer to Table 612, which is a structure of pointers 613. The pointers 613 and 604 may reference any memory location within the system RAM 601.

System programmers should be advised that optimal performance is realized when Table 612 and Table 614 are located in the main system RAM 601 (e.g., eDRAM 108 in the case of a smart phone).

Table 612 is a fixed structure of 32 entries. Each entry represents one of the 32 IPCM DMA data channels and contains a 32-bit pointer. The IPCM 116 uses the pointer in Table 602 to find Table 612. Table 612 must be present in its entirety even when all 32 channels are not used. The IPCM 116 will not use pointer entries for channels which are not enabled (e.g., a bit set in the channel enable register 606).

It is intended that the value of the pointer(s) in Tables 602 and 612 remain "static" for the duration of the IPCM operating session.

Table 614 contains the operating command and status registers, a buffer size descriptor (e.g. count field 619) and a pointer 616 to the physical buffer 618 itself. The dedicated table 614 must be present for each of the 32 channels and is implemented as a Linked-List structure. This technique permits one or more buffers 618 to be allocated to a single DMA channel and thereby

enables a variety of implementation schemes such as buffer swapping and daisy chaining.

In operation, the IPCM 116 will read the command register element of the first buffer described in the list and rise that buffer 618 until it is filled to capacity or closed by some event. The IPCM 116 will refer back to the Linked-List and open the next buffer in the list if one is declared. When the IPCM 116 reaches the end of the list, all buffers 618 will have been filled, or closed, arid the specific IPCM channel referenced by Table 614 will be shutdown. The channel may be restarted by the MCU updating the contents of Table 614 for the respective channel followed by asserting the appropriate channel enable bit in the Channel Enable Register 606 located in the IPCM.

A variety of operating scenarios are possible using the nodes provided. Buffers 618 may be alternately swapped, daisy chained, or operated in modulo mode. Interrupts may be generated to the Host MCU or DSP when a specific buffer is filled, or closed, by configuring the appropriate control bit within the Mode Register.

The Mode field 620 within the structure of Table 614 contains a control bit to serve as a semaphore and referred to as the "D" bit. When set the buffer is owned by the IPCM and the count, pointer, status, and control fields must not be changed by the MCU. When D=0, the MCU owns that particular buffer and the IPCM will not attempt to write into any of the control fields or buffer space.

Note that the channel interface to Quartz DSP is similar to that of the host MCU.

Host Processor Programming Model

The host processor 106 (simply referred to as the host) controls the IPCM 116 by means of several interface registers. They are all accessed with 0

wait-state on the ip bus interface except the once command register (ONCE_CMD) that requires 1 wait-state and the CHENENBL RAM that also requires 1 wait-state. They are all clocked with the IPCM clock (which means the Host must ensure the IPCM clock is running when it wants to access any register).

Regarding read & write authorized sizes; any read puts the 32-bit contents of the register on the bus regardless of the read size (i.e., byte enables are ignored); any write updates the contents of the register according to the required size (i.e., byte enables are used to allow writing of the corresponding byte from the bus to the register) except when writing to the CHNENBL RAM that always stores the full size word on the bus (32-bit) into the RAM, regardless of the size.

Following are the registers of the host processor used in controlling the IPCM 116 in accordance with one embodiment of the invention.

COPTR (Channel 0 Pointer). This register contains the 32-bit address, in host memory, of the array of channel control blocks starting with that for channel 0 (the control channel). The host has a read/write access, the IPCM has a read-only access. On reset, this register will be all zeros. This register should be initialized by the host before it enables a channel (e.g. channel 0).

INTR- Channel Interrupts. This register contains the 32 HI[i] bits. If any bit is set, it will cause an interrupt to the host. This register is a "write-ones" register to the host. When the host sets a bit in this register, the corresponding HI[i] bit is cleared. The interrupt service routine should clear individual channel bits when their interrupts are serviced, failure to do so will cause continuous interrupts. The IPCM is responsible for setting the HI[i] bit corresponding to the current channel when the

corresponding done instruction is executed.

STOP/STAT- Channel Stop/Channel Status. This 32-bit register has one bit for each channel. This register is a "write-ones" register to the host. When the host writes I in bit i of this register, it clears the HE[i] and START[i] bits. Reading this resister yields the current state of the HE[i] bits.

START- Channel Start. This 32-hit register has one bit for each channel. This register is a "write-ones" register to the host. Neither START[i] bit can be set while the corresponding HE[i] bit is cleared. When the host tries to set the START[i] bit by writing a one, if the corresponding HE[i] bit is clear, the bit in the START[i] register will remain cleared and the HE[i] bit will be set. If the corresponding HE[i] bit was already set, the START[i] bit will be set. The next time the IPCM channel i attempts to clear the HE[i] bit by means of a done instruction, the bit in the START[i] register will be cleared and the HE[i] bit will take the old value of the START[i] bit. Reading this register yields the current state of the START [i] bits. That mechanism allows the Host to pipe-line two START commands per channel.

EVTOVR- Channel Event Override. This register contains the 32 EO[i] bits. A bit set in this register causes the IPCM to ignore events when scheduling the corresponding channel. Writing in this register overwrites the previous value.

DSPOVR- Channel DSP Override. This register contains the 32 DO[i] bits. A bit set in this register causes the IPCM to ignore DSP enable when scheduling the corresponding channel.

HOSTOVR- Channel HOST Override. This register contains the 32 HO[i] bits. A bit set in this register causes the IPCM to ignore HOST enable when scheduling the corresponding channel.

EVTPEND- Channel Event Pending. This register contains the 32 EP[i] bits. Reading this register allows the host to determine which charnels have events pending. Setting a bit in this register causes the IPCM to reevaluate scheduling as if a peripheral event destined for this channel had occurred (this a `write-ones' mechanism: writing an `0' does not clear the corresponding bit). This is useful for starting up channels, so that initialization is done before awaiting the first event. The task scheduler can also sets bits in the EVTPEND register, according to the received events. The EP[i] bit may be cleared by the done instruction when running channel I script.

DSPENMBL- Channel DSP Enable. This register contains the 32 DE[i] bits (from the DSP Control module or control registers). This register enables the host to determine which channels the DSP has enabled.

RESET- Reset Register. This register contains two control bits: (1) the reset bit (RESET[0]), when set causes the IPCM to be held in software reset. The reset has to be held some cycles before it cam be released. Actually a counter is used to keep the soft reset (soft_rstB low) active during these cycles. The soft_rstB signal is driven low as the reset bit is set. The reset bit is cleared by the IPCM. And (2) The resched bit (REST[i]), when set, forces the IPCM to reschedule. as if a script had executed a done instruction. This allows the host to recover from a runaway script on a channel by clearing its HE[i] bit via the STOP register and then forcing a reschedule via the RESCHED bit of the RESET register. When the context switch starts. the resched bit is cleared by the IPCM.

EVTERR- Event Error Register. This register is used by the IPCM to warn the host when an incoming event was detected and it triggers a channel that is already pending or being serviced. That probably means there is

an overflow of data for that channel. This is a "write-ones" register for the scheduler; it is only able to set the flags; the flags are cleared when the register is read by the Host or during IPCM reset.

Any EVTERR[i] bit is set when an event that triggers channel i has been received through the event input pins and the EP[i] bit is already set; the EVTERR[i] bit is unaffected if the Host tries to set EP[i] bit whereas that EP[i] bit is already set.

This register is NOT the same as the DSP EVTERR: when the Host reads and clears it, the same information is still available in the DSP register.

INTRMASK- Channel Interrupt Mask Flags. This register contains 32 interrupt generation mask bits. If bit INTRMASK[i] is set, the HI[i] bit is set and an interrupt is sent to the Host whenever an event error is detected on channel i (i.e., EVTERR[i] is set).

PSW- Scheduler Status. This is a 16-bit register with the following status information: [4:0]CCR (current channel register); [7:5]CCP (current channel priority); [12:8]NCR (next channel register); and [15:13]NCP (next channel priority).

EVTERRDBG- Event Error Register for Debug. This register is the same as EVTERR except reading it does not change its contents (i.e., it is not cleared); that address is meant to be used in debug mode: the MCU OnCE may check that register value without modifying it.

ONCE_ENB- OnCE Enable. That one-bit register selects the OnCE control source; when cleared (0), the JTAG controls the OnCE; when set (1), the Host controls the OnCE through the registers described below. After reset, the Once enable bit is cleared (JTAG controls).

ONCE_CMD- OnCE Command Register (1 wait state). Writing to that register will cause the OnCE to execute the written command; when needed, the ONCE_DATA and

ONCE_INSTR registers should be loaded with the correct value before writing the command to that register.

CSWADDR- Context Switch Address. A 15-bit register that contains the context switch address (bits 13-0) and an enable bit (14); when the enable bit is set, the context switch routine is assumed to start at the address contained in bits 13-0. The reset value of that register is 0 for the enable bit and decimal 32 for the context switch address.

ILLINSTADDR- Illegal Instruction Trap Address. Bits 13-0 of this register contain the address where the IPCM jumps when an illegal instruction is executed; it is 0x0001 at reset.

CHN0ADDR- Channel 0 Address. This 13-bit register is used by the boot code or the IPCM: after reset, it points to the standard boot routine in ROM (channel 0 routine); by changing that address, the user has the ability to perform a boot sequence with his own routine. The very first instructions of the boot code fetch the contents of that register (it is also mapped in the IPCM memory space) and jump to the given address. The reset value is 0x0050 (decimal 80).

CHNENBL- Channel Enable RAM. The Host Control module contains a 32x32 channel enable RAM. This channel enable RAM contains the event/channel correspondence map. Any event can trigger any possible combination of channels according to the contents of that RAM.

CHNPRI- Channel Priority Registers. This set of 32 registers contains the priority of every channel; that number is comprised between 1 and 7. 0 is a reserved value used by the IPCM hardware to detect when no channels are pending.

DSP Programming Model

The DSP 102 has some limited, compared to the host processor 106 (i.e. host), control over the IPCM 116

via several interface registers. Each register occupies two 16-bit words to accommodate all 32 channels. All registers operate in the same manner as there host processor counterparts. The CEVTOVR and CDSPOVR registers are not implemented in the DSP interface.

All addresses, such as the C0PTR, are comprised of one 32-bit word that may address any location within the Quartz memory space (4 Gbytes).

All registers are clocked with the IPCM clock (which means the Host must ensure the IPCM clock is running when the DSP is supposed to access those registers).

Regarding read & write authorized sizes: any read puts the 32-bit contents of the register on the bus, regardless of the read size (i.e., byte enables are ignored); any write updates the contents of the register according to the required size (i.e., byte enables are used to allow writing of the corresponding byte from the bus to the register).

The following are the registers of the DSP used in controlling the IPCM 116 in accordance with one embodiment of the invention.

C0PRT- Channel 0 Pointer. This register contains the 32-bit address, in DSP memory, of the array of channel control blocks starting with CCB for channel 0 (the control channel). The DSP has a read/write access, the IPCM has a read-only access. On reset, this register will be all zeros. This register should be initialized by the DSP before any channels are enabled.

INTR- Channel Interrupts. This register contains the 32 DI[i] bits. If any bit is set, it will cause an interrupt to the host. This register is a "write-ones" register to the host. When the host sets a bit in this register the corresponding DI[i] bit is cleared. The interrupt service routine should clear

individual channel bits when their interrupts are serviced, failure to do so will cause continuous interrupts. The IPCM is responsible for setting the DI[i] bit corresponding to the current channel when the corresponding done instruction is executed.

STOP/STAT- Channel Stop/Channel Status. This 32-bit register has one bit for every channel. This register is a "write-ones" register to the DSP. When the DSP writes a 1 in bit I of this register, it clears the corresponding DE[i] and START[i] bits. Reading this register yields the current state of the DE[i] bits.

START- Channel Start. This 32-bit register has one bit for each channel. This register is a "write-ones" register to the DSP. Neither START[i] bit can be set while the corresponding DE[i] bit is cleared. When the DSP tries to set the START[i] bit by writing a one, if the corresponding DE[i] bit is clear, the bit in the START [i] register will remain cleared and the DE[i] bit will be set. If the correspondent DE[i] bit was already set, the START [i] bit will be set. The next time the IPCM channel I attempts to clear the DE[i] bit by means of a done instruction, the bit in the START [i] register will be cleared and the DE[i] bit will take the old value of the START[i] bit. Reading this register yields the current state of the START[i] bits. That mechanism allows the DSP to pipe-line two START commands per channel.

EVTERR- Event Error Register. This register is used by the IPCM to warn the DSP when an incoming event was detected and it triggers a channel that is already pending or being serviced. That probably means there is an overflow of data for that channel. This is a "write-ones" register for the scheduler. It is only able to set the flags; the flags are cleared when the register is read by the DSP or during IPCM reset.

Any EVTERR[i] bit is set when an event that

triggers channel i has been received through the event input pins and the EP[i] bit is already set; the EVTERR[i] bit is unaffected if the Host tries to set the EP[i] bit whereas that EP[i] bit is already set.

This register is not the same as the Host EVTERR. When the DSP reads and clears it, the same information is still available in the Host register.

INTRMASK- Channel Interrupt Mask Flags. This register contains 32 interrupt generation mask bits. If bit INTRMASK[i] is set, the DI[i] bit is set and an interrupt is sent to the DSP whenever an event error is detected on channel i (i.e., EVTERR[i] is set).

PSW- Scheduler Status. This is a 16-bit register with the following status information: [4:0]CCR (current channel register); [7:5]CCP (current channel priority); [12:8]NCR (next channel register); and [15:13] NCP (nest channel priority).

EVTERRDBG- Event Error Register for Debug. This register is the same as EVTERR except reading it does not change its contents (i.e. it is not cleared); that address is meant to be used in debug mode. The DSP OnCE may check that register value without modifying it.

IPCM Programming Model

Each programmable DMA data channel has eight general purpose registers of 32-bits for use by scripts. General register 0 has a dedicated function for the loop instruction, but otherwise can be used for any purpose.

Functional Unit State. Each channel context has some state that is part of the functional units. The specific allocation of this state is part of the functional unit definition. This state must be saved/restored on context switches.

Program Counter Register (PC). The PC is 14 bits. Since instructions are 16-bits in width and all memory in the IPCM is 32-bits in width, the low order bit

of the PC selects which half of the 32-bit word contains the current instruction. A low order bit of zero selects the most significant half of the word (i.e. big-endian).

Flags. Each channel has 4 flags: The T bit reflects the status of some arithmetic and test instructions. It is set when the result of an addition or a subtraction is zero and cleared otherwise. It is also the copy of the tested bits. Finally it can also be set when the loop counter ((GReg0) reaches zero; when the last instruction of the hardware loop is an operation that can modify the T flag, its effect on T is discarded and replaced by the GReg0 Status.

Two additional bits, SF and DF, are used to indicate error conditions resulting from loading data sources and storing to destinations, respectively. Access errors set these bits, and successful transactions clear them. They can also be cleared by specific instructions (CLRF and LOOP. The SF (source fault) is updated by loads LD and LDF; the DF (destination fault) is updated by stores ST and STF.

Access errors are caused by several conditions: writing to the ROM, writing to read-only memory mapped register. accessing art unmapped address or any transfer error received by a peripheral when it is accessed.

The SF and DF flags have a major impact on the behavior of the hardware loop: if SF or DF is set when starting a hardware loop and it is not masked by the LOOP instruction, the loop body will not be executed; now, inside the loop body, if a load or store sets the corresponding SF or DF flag, the loop exits immediately. Testing the status of the T flag at the end of the loop (as well as testing both SF and DF) tells If the loop exited abnormally as any anticipated exit prevents Greg0 from reaching the zero value and thus setting the T flag. This is also valid if the fault occurs at the last instruction of the last loop.

The last flag is the loop mode flag, LM, which indicates when the processor is currently operating in loop mode. It is set by the LOOP instruction and is cleared after execution of the last instruction of the last loop.

Return Program Counter (RPC). The RPC is 4 bits. It is set by the jump to subroutine instructions and used by the return from subroutine instruction. Instructions are available to transfer its contents to and from a general register.

Loop Mode Start Program Counter (SPC). The SPC is 14 bits. It is set by the loop instruction to the location immediately following it.

Loop Mode End Program Counter (EPC). The EPC is 14 bits. It is set by the loop instruction to the location of the next instruction after the loop.

Context Switching. Each channel has a separate context consisting of the 8 general purpose registers and additional context representing the state of the functional units. The active registers and functional units contain the context of the active channel. The context of in inactive channels are stored in IPCM RAM which is part or the IPCM address space. A context switch stores the active registers into the context area of the old channel and loads the new context from the context area of the new channel. It exactly requires 47 IPCM cycles to complete.

It is possible to define a custom context switch routine. The user has to store it wherever possible in RAM and its start address must be written in the CSWADDR control register via the MCORE. With that option it is not possible to achieve a similar cycle count as the built-in routine (i.e., 47 cycles) as all loads and stores will require 2 cycles to complete instead of 1 cycle in the ROM routine (1 cycle to perform the load/store plus 1 cycle to fetch the next

instruction: both accesses use the RAM, which means they cannot be done in parallel).

Memory Mapped Registers. The IPCM core has access to several registers through the system bus.

Host Channel 0 Pointer (MC0PTR). Contains the address, in the MCU memory space, of the initial IPCM context and scripts, that are loaded by the IPCM boot script which is running on channel 0. This is a read-only register.

DSP Channel 0 Pointer (DC0PTR). Contains the address, in the DSP memory space, of the initial IPCM context and scripts, that are loaded by the IPCM boot script which is running on channel 0. It is unused for now. This is a read-only register.

Current Channel Register (CCR). Contains the 5-bit priority of the channel whose context is installed. This is a read-only register.

Current Channel Priority (CCPR). Contains the 3-bit priority of the channel whose context is installed. This is a read-only register.

Highest Pending Channel Register (HPCR). Contains the decoded 32-bit number of the channel the task scheduler has selected to run next. A bit is set to "1" at position or channel selected (e.g., if HPCR contains value 0x04000000, channel 26 is the next channel selected by the scheduler). This is a read-only register.

Highest Pending Priority (HDPR). Contains the 3-bit priority of the channel the scheduler has selected to run next. This is a read-only register.

Current Channel Pointer (CCPTR). Contains the start address of the context data for the current channel: its value is CONTEXT_BASE + 20*CCR (CONTEXT_BASE = 0x0800); this is a read-only register.

CHN0ADDR. Contains the address of the channel

0 routine programmed by the MCORE; it is loaded into a General register at the very start of the boot and the IPCM jumps to the address it contains. By default, it points to the standard boot routine in ROM.

Address Space. The IPCM has two internal busses: the Instruction bus used to read instructions from the memory; and the data bus used to access the same memories as those visible on the instruction bus, plus some memory mapped registers (scheduler status and OnCE registers) and 5 peripheral registers (USB, UART1, UART3, MMC & Video SAP).

Instruction Memory Map. It is based on a 14-bit address bus and a 16-bit data (instruction) bus; instructions are fetched from either program ROM or program RAM. An IPCM script is able to change the contents of the program RAM that is also visible from the data bus.

The first two instruction locations (at 0 and 1) are special. Location 0 is where the PC is set on reset. Location 1 is where the PC is set upon the execution of an illegal instruction. It is expected that both of these locations will contain a jmp to handler routines.

Data Memory Map. All of the data accessible to IPCM scripts make tip the data memory space or the IPCM. This address space has several components: ROM, RAM, peripheral registers, and scheduler registers (CCR, HPCR and CCPTR) and OnCE registers. IPCM scripts can read and write to the context RAM, data RAM and peripheral registers.

The address range is 16 bits and the data width os 32 bits; however, when accessing peripheral registers (USB, etc.), the data width may be different; in that case, during a write, the unused part of the 32-bit data to write is ignored by the peripheral; during a read, the missing part of the 32-bit read data is replaced by `0's.

IPCM Initialization

After hardware reset, the IPCM 116, the program RAM, context RAM, and data RAM have unpredictable contents. The active register set is assigned to channel 0 and the PC is initialized to all zeros. However, since the channel enable register is all zeros, there are no active channels and the IPCM is halted.

To start up the IPCM, the host processor 106 (i.e. host) first creates some channel control blocks in host memory for the control channel (channel 0) and then initializes the channel 0 pointer register to the address of the first control block. It then sets bit 0 (corresponding to channel 0) in the channel enable register.

Upon being enabled, the IPCM 116 then begins executing the script located at address 0 in the program ROM for channel 0. This ROM 308 script will read the channel 0 pointer register and, using the address contained therein, begin fetching (using DMA) the first channel control block. If the block contains a valid command, it interprets the command (which will normally be to download something from host to IPCM memory) and proceeds to implement the command and move on to the next control block. This continues until an invalid channel command is reached, at which time the script will halt, awaiting the host to re-enable the channel again.

There are also two means to make the IPCM boot on a user-defined script. First, by using the OnCE (either via its Jtag interface of its MCORE interface) to download any code in the IPCM RAM and force the IPCM to boot on that code; second, by using the CHN0ADDR register in the Host programming model; the IPCM boot code fetches the contents of that register and jumps to the given address.

The execution of an IPCM script depends on both the instructions that make up the script and the data

context upon which it operates. Both must be initialized before the script is allowed to execute. Each of the 32 channels has a separate data context, but may share scripts and locations in data RAM.

The host manages the space in program RAM and data RAM. It also manages the assignment of IPCM channels to the device drivers that need them. Channels are initialized by the host by using channel 0 to download any required scripts and data values and the channels initial context. The context contains all the initial values of the registers, including the PC. Then the host 106 enables the channel and the channel becomes active and begins fetching and executing instructions from its script.

Refer to attached Appendix A entitled INSTRUCTION SET FOR RISC PROCESSOR CORE OF IPCM, which describes a complete set of preferred instructions for use in the IPCM 116 in accordance with one embodiment of the present invention.

The following further describes the DSP DMA unit and the host processor DMA unit corresponding to the embodiment employing the instruction set of Appendix A. functional units.

The functional unit instructions cause an 8-bit code, found in the low 8-bits of the instruction, to be asserted on the functional unit control bus. Some of these bits are used to select one of several functional units. In order to establish a programming convention, we will assume the selection bits are some number of the most significant bits of the 8-bit code. Furthermore, some number or the least significant bits will be decoded by a given functional unit to establish the type of operation to perform.

For the host DMA unit 312, the DMA instructions control the DMA state machine and may cause a DMA cycle on the associated memory bus. There are three registers

associated with the host DMA unit, an address register (MA), a data buffer (MD) and a state machine register (MS).

The address register (MA) contains the pointer into DMA memory associated with the next data transfer. It has byte granularity. Reading the register with the ldf instruction (i.e. read) has no side effects. Writing the address register may have side effects. If there is value write data in the buffer, and the address is changed, the write data will be flushed (i.e., a DMA write cycle will be issued). If the prefetch bit is set and if there are no valid write data, a DMA read cycle will be issued with the new address.

As data is transferred to or from the data buffer, the address register is incremented by the number of bytes transferred/ Of the address increments across a 32-bit word boundary any valid write data in the buffer will be flushed.

In the data buffer register (MD), a DMA cycle is not always associated with a stf instruction which loads the write buffer, the instruction may just load a subunit of transfer into the buffer register as it accumulates bytes which will be later used in full size memory transfers. The DMA unit keeps state as to which bytes are valid and does the correct shifting and insertion of new data. The instruction that loads the write buffer can conditionally cause the resulting buffer to be flushed, causing a DMA write cycle, even if the buffer is not entirely filled.

A ldf instruction that reads the data buffer, may cause a DMA cycle if the data has not already been fetched. Each read transfer can conditionally cause a prefetch, if all the bytes in the buffer have been transferred.

Writes and reads of the data buffer may cause destination or source faults, respectively. As the MLB

does not support 24-bit accesses, the Host DMA triggers an error when a 3-byte access is requested on the MLB. That only no occurs when a flush or a fetch is requested; the internal MD register can have a 3-byte data at any moment as far as no external access is performed. The eDRAM supports 3-byte accesses.

There is the special case of the flush: using byte accesses to MD, it is possible to have 3 active bytes and request a flush; the 3 active bytes in MD will be correctly written to the eDRAM; but the access will cause an error if the write is done on the MLB.

The state register (MS) contains the DMA state-machine value. It is not meant to be accessed by the user in normal mode. In fact, as context switches may occur while the DMA is in any state, it is necessary to save that state, which is done by the context switch routine.

The Prefetch and Flush management allows the IPCM RISC machine to go on while a DMA access is performed. When the RISC Core requires a prefetch (p=1) or an auto-flush (f=0) to the Host DMA, it will receive an immediate transfer acknowledge before the DMA has finished the external access; which allows the RISC Core to do other things like accessing another DMA machine.

However, the user must be aware of the inherent limits of that mechanism: as far as the DMA has not a FIFO stack to store commands, if a prefetch/auto-flash command is issued, whereas the DMA has not finished its previous access, the transfer acknowledge will be delayed until the preceding access is over.

Another point is the management of errors: as the DMA immediately sends an acknowledge to the RISC Core, it assumes no error will occur (except if it detect the access is forbidden like a 24-bit access to the MLB). If an error occurs, it will be flagged (transfer error acknowledge) for the following DMA access.

That should not be a problem if the DMA is used properly. The prefetch/auto-flush feature is meant to be used in hardware loops and a last access with no prefetch (p=0) or a forced flush (f=1) should be performed after the hardware loop: that access will gather any remaining error (its own as well as an error from the previous prefetch or auto-flush access).

The DSP DMA Unit 310 is functionally identical to the host DMA unit 312 with minor restrictions. It allows 32-bit and 16-bit transfers to the Quartz DMA.

The DMA instructions control the DMA state machine and may cause a DMA cycle on the associated memory bus. There are three registers associated with the host DMA unit, an address register (DA), a data buffer (DD) and a state machine register (DS).

The address register (DA) contains the pointer into DMA memory associated 'with the next data transfer. It has byte granularity. Reading the register with the ldf instruction has no side effects. Writing the address register may have side effects. If there is valid write data in the buffer, and the address is changed, the write data will be flushed (i.e., a DMA write cycle will be issued). If the prefetch bit is set and if there are no valid write data, a DMA read cycle will be issued with the new address.

As data is transferred to or from the data buffer, the address register is incremented by the number of bytes transferred. If the address increments across a 32-bit word boundary any valid write data in the buffer will be flushed.

In the data buffer register (DD), a DMA cycle is not always associated with a DMA write instruction which loads the write buffer. The instruction may just load a subunit of transfer into the buffer register as it accumulates bytes which will be later used in full size memory transfers. The DMA status keeps state as to which

bytes are valid and does the correct shifting and insertion of new data. The instruction that loads the write buffer can conditionally cause the resulting buffer to be flushed, causing a DMA write cycle, even if the buffer is not entirely filled.

A DMA read instruction that reads the data buffer, may cause a DMA cycle if the data has not already been fetched. Each read transfer can conditionally cause a prefetch, if all the bytes in the buffer have been transferred.

Writes and reads of the data buffer may cause destination or source faults, respectively. As the Quartz DMA does not support byte accesses, the DSP DMA detects any unauthorized access size and triggers an error accordingly. Unauthorized sizes are 1 byte and 3 bytes. That only occurs when a flush or a fetch is requested. The internal DD register can have a I-byte or a 3-byte data at any moment as far as no external access is performed.

The state register (DS) consists of 00110000 32-bit read with no side effect.

The Prefetch and Flush management allows the IPCM RISC processor to go on while a DMA access is performed. When the RISC Core requires a prefetch (p=1) or an automatic flush (f= 0) to the Host DMA, it will receive an immediate transfer acknowledge before the DMA has finished the external access; which allows the RISC Core to do other things like accessing another DMA machine.

However, the user must be aware of the inherent limits of that mechanism: as far as the DMA has not a FIFO stack to store commands, if a prefetch/auto-flush command is issued whereas the DMA has not finished its previous access, the transfer acknowledge will be delayed until the preceding access is over.

Another point is the management of errors: as the DMA immediately sends an acknowledge to the RISC Core, it assumes no error will occur (except if it detects the access is forbidden like a 24-bit access to the MLB). If an error occur, it will be flagged (transfer error acknowledge) for the following DMA access.

That should not be a problem if the DMA is used properly: the prefetch/auto-flush feature is meant to be used in hardware loops and a last access with no prefetch (p=0) or forced flush (f=1) should be performed after the hardware loop: that access will gather any remaining error (its own as well as an error from the previous prefetch or auto-flush access).

Programming Conventions

Much of the programming model as seen by the host or DSP is not mandated by the hardware architecture of the IPCM, but rather by the scripts that run on the IPCM. Some of these scripts will be in ROM so the conventions they impose are not easily changeable.

On the host processor side, There are 32 channel control blocks (CCBs) in a array whose base address is specified in the COPTR. Each control block consists of four 32-bit words. The first word will contain status which is currently undefined. The second word contains a pointer to the base of an array of buffer descriptors (Bds). The third word contains a pointer to the current BD. The fourth word is currently unused.

The contents of a channel control block may only be changed by the host when the channel is not running.

The host buffer descriptors (refer to FIG. 6) form an array of programmable size, the last buffer descriptor is marked as such. The array of buffer descriptors is treated as a ring, with some logically

contiguous portion owned by the host, and the remainder by the IPCM. A status bit indicates the ownership of each buffer descriptor.

When a buffer descriptor changes ownership from the host to the IPCM, the count field indicates how much data is to be transmitted or the size of the receive buffer. When ownership reverts back to the host, the count indicates how much data was transmitted or received.

Channel 0 Commands. The COMMAND field of a buffer descriptor contains an 8-bit command code used to communicate between the host and the IPCM. Currently the channel 0 script recognizes only the following commands: (1) Set the IPCM address to be used in subsequent commands to the value contained in the buffer address field; (2) Copy from the host memory at buffer address to the IPCM memory; (3) Copy to the host memory at buffer address from the IPCM memory; (4) Copy from the host memory to the channel context of the channel number in the high 5 bits; and (5) Copy to the host memory from the channel context of the channel number in the high 5 bits.

On the DSP side, there are 32 channel control blocks (CCBs) in a array whose base address is specified in the COPCR. The CCB for channel 0 is not used. Each channel control block consists of eight 16-bit words/ The first two words will contain status which is currently undefined. The second two words contain a pointer to the base of an array of buffer descriptors (Bds). The third two words contain a pointer to the current BD. The fourth two words are currently unused.

DYNAMIC MEMORY REFRESH METHODS

Referring briefly back to FIG. 2, the processor platform is illustrated including the refresh controller responsible for refreshing the dynamic random access

memory, i.e. eDRAM. The refresh controller 220 is coupled to the eDRAM 108 via the r bus 240 through the data path select 218, which acts to multiplex access to the eDRAM from other competing devices, such as the IPCM 116, the host processor 106, and the LCD controller 204.

As is known in the art, dynamic memory (DRAM or eDRAM) is required to be provided DC power and refreshed periodically so that the data stored in the memory is retained. In contrast, static memory (e.g. SRAM) does not need to be refreshed as long as a constant power supply is coupled thereto. The refresh controller 220 of one embodiment of the processor platform 100 is designed to conserve as much power as possible, which for handheld applications is advantageous in extending battery life.

The refresh controller 220 has two basic modes of operation: an active mode and a data retention mode. The active mode is invoked during normal operation of the eDRAM 108. The active mode is used anytime the product incorporating the processor platform 100 is operated by the end-user. The active mode uses a "cycle stealing" distributed refresh method as known in the art of discrete DRAM implementations.

Data retention mode is invoked when the product incorporating the processor platform 100 is not operational and it is desired to retain the contents of the eDRAM 108. This data retention mode has three operating modes: (1) distributed refresh method, (2) a "selective refresh" method and (3) a "temperature compensated" refresh method.

The distributed refresh method as known in the art is identical to the active mode of operation in that the memory is refreshed periodically, whereas the selective refresh and the temperature compensated refresh may be used individually or together in order to reduce the amount of power required to retain data in the eDRAM memory array or may be used together to achieve a further

overall reduction in the power consumed to retain data, which results in a longer battery life in which is particularly important to handheld applications.

Even though the active mode and distributed refresh technique are well known in the art, the refresh controller 220 advantageously uses a low frequency clock source from a time of day module operating at 32 kHz, although in other embodiments any clock known in the art may be used. Thus, in preferred embodiments, the clock input 252 to the refresh controller 220 is the time of day clock required for use in the product incorporating the processor platform 100. This 32 kHz clock input 252 is multiplied (x2) within the refresh controller 220 and used to activate the refresh cycle. As is known in the art, a counter is provided within the refresh controller 220 having 'n' number of states where 'n' is equal to the total number of rows in the array. The counter is incremented modulo 'n' with each clock transition. "Sense amplifiers", as known in the art, perform the actual refresh of the selected bit cells within the rows.

Advantageously, the refresh controller 220 uses the already provided time of day clock as the clock input; thus, eliminating the need for a separate dedicated clock to run the refresh controller 220 as is done conventionally. Employing a separate dedicated clock for the refresh controller 220 further adds to the power consumed by the system when the power is off.

Referring next to FIG. 7, The selective refresh method advantageously reduces the numbers of rows in the memory (e.g. eDRAM 108) to be refreshed in order to reduce power consumption at the expense of reducing the number of memory cells that will be retained. For example, if the memory (i.e. DRAM) is divisible into multiple portions, one or more of the multiple portions of the memory may be refreshed without refreshing all of the multiple portions. Specifically, the multiple

portions may be rows of memory in an array. Thus, the contents of the rows of the memory being refreshed will be saved, while the contents of the data held in memory rows not being refreshed will be lost. For example, while the device is being shut down by the user (e.g. turned off), the host processor 106 makes a determination of which rows within the memory, e.g. eDRAM 108, need to be saved and which portions (e.g. rows) do not need to be saved (Step 702). Next, the host processor 106 sends a control signal to the refresh controller 220 instructing which rows or portions of the host processor memory is desired to be saved (Step 704). The refresh controller 220 is configured to refresh only the identified rows or portions within memory for the duration of time that the power is off. Then, the refresh controller 220 accesses the host processor memory (e.g. eDRAM 108 through the data path select 218) and refreshes the configured rows (Step 706). Thus, less power is consumed in the refresh of the memory since only part of the memory is being refreshed, instead of the entire memory being refreshed. In this embodiment, this provides quite a savings in power since the eDRAM 108 is large and typically less than the entire eDRAM 108 needs to be saved.

The selective refresh method implemented by the refresh controller 220 is described as follows. Given that there are n rows within the memory array, e.g. eDRAM 108, define j as the total number of rows in the memory array to be refreshed such that j is a number between 0 and n. The algorithm then becomes:

i=0
Refresh $Row_i$
i=i+1 modulo j.

In other words, the refresh controller 220 modulo `n' counter is reduced to a modulo (n-a) counter where `a' is the number of rows to be omitted from the refresh activity.

Referring next to FIG. 8, a flowchart 800 is shown of the steps of the refresh controller 220 of FIG. 2 in performing the temperature compensated method of memory refresh. The temperature compensated method of memory refresh adjusts the periodicity of the refresh activity based on ambient temperature of the product. At room temperature and below, this can save a considerable amount of power, which is important to increasing battery life in battery operated handheld devices.

Generally, the higher the temperature, the more often a given memory, e.g. eDRAM, is required to be refreshed. Prior art approaches design the refresh rate of a refreshing unit around "worst case" scenarios in which the product incorporating the DRAM would be at an unusually high default temperature. As such, for most of the time, the memory is actually "over-refreshed". However, in size and power conscious applications, such as for use in the processor platform 100 used as a multimedia wireless handheld device where power consumption is desired to be minimized, such "over-refreshing" may lead to a waste of power in operating the refresh controller more than necessary.

Thus, in contrast to conventional refresh controllers, the default temperature assumed for the refresh controller 220 is very low, such that the default refresh rate is less (i.e. the time in between refreshes is longer), often resulting in a savings of power. However, the temperature of the product may be at this low temperature or below, but at other times the ambient temperature of the product will be above this low refresh rate temperature. Unless the refresh rate is compensated for higher temperatures, the data retained in the DRAM will be lost in between refresh cycles. Thus, the temperature compensated method of refreshing measures the ambient temperature of the product in order to determine if the refresh rate needs to be increased or if it can remain at a slower refresh cycle in order to save power.

When the ambient temperature of the product is high, the time between refresh cycles is decreased.

Specifically, the temperature compensated refresh method is described below. First, given a digital timer circuit that is clocked with an accuracy of +/-100ppm and having a resolution of at least .1 second. The timer shall be capable of measuring time using a suitable clock signal and signaling an "event" after a pre-programmed time has elapsed. The timer shall have a register which is set to an integer value and is used to represent a pre-programmed value elapsed time to be measured. Also given an electronic digital thermometer circuit providing an indication range of x to y degrees with an accuracy of +/"j" degrees and further given a translation table consisting of a non-volatile memory array (e.g. a pre-programmed ROM) of "n" elements with each element containing an integer number herein referred to as "count". The bounds of the array (i.e. number of elements) are determined by the following equation: n=y-x, where n is the number of array elements and x and y upper and lower temperature range limits.

The absolute value and range of "count" must be compatible with the absolute value and range of the "count" register that is supported in the digital timer. Each "count" entry is assigned to occupy one element within the array. The entries of "count" are ordered in the array in a linear ascending manner such that the first element in the array corresponds to the lowest measured temperature (i.e. x) and the last element in the array corresponds to the highest measured temperature (i.e. y).

The value assigned to each count entry is made from empirical data taken from actual samples of the actual DRAM memory device intended to be used. As an alternative, the values for "count" may be determined through computer simulation methods of the leakage characteristics of the transistors used in the memory

array as a function of changes in ambient temperature. In either case, the value of "count" is to represent the minimum refresh rate necessary to maintain data integrity at a specific temperature within the range of x to y degrees. Each element in the array is to represent an incremental change in ambient temperature staring at temperature x and incrementing to temperature y. The contents of each element is to represent the minimum necessary refresh rate to maintain data integrity. Additional tolerance must be given to accommodate digital temperature accuracy "j" as well as memory operating voltage tolerance and expected unit to unit variation. It is expected that while the elements of the array are ordered in linear fashion with respect to temperature, the values of "count" may represent an exponential characteristics.

The temperature compensated refresh method operating procedure begins by refreshing all rows in the memory array (Step 802). Note that rows to be refreshed may be governed by "selective refresh method" described above with reference to FIG. 7, such that Step 802 may be refreshing less than all of the rows (portions) of memory in the memory array. Next, an ambient temperature is measured (Step 804) using digital thermometer. Next, the ambient temperature measurement is translated to a count value using a look-up table (Step 806). The lookup table represents various refresh rates at different temperatures determined through simulation and empirical data. If the measured temperature is out of bounds for the lookup table then the first element, in the case exceeding the lower boundary "x", or the last element, in the case of exceeding the higher boundary "y", should be chosen. The count value is applied to the digital timer circuit (Step 808). Once the timer "event" has expired, i.e. the digital timer circuit expires (Step 810), refresh all rows; thus, repeating Steps 802 through 810. All steps are repeated indefinitely until the product is

turned on (Step 812); thus, active refresh mode is re-entered (Step 814).

Thus, by employing the temperature compensated refresh method of memory refresh, power is conserved, especially at or below room temperature operating environments, in comparison to a standard refresh operation, e.g. the distributed refresh method.

The "temperature compensated mode" together with the "selective refresh mode" can advantageously be operated simultaneously in order to consume the lowest possible data retention power. The operating modes of the refresh controller are configured by the host processor via signaling overt the h bus 232 prior to product shut down. The refresh controller 220 automatically reverts to the "Normal" refresh mode of operation (e.g. distributed refresh) when the host processor 106 re-enters the active state, i.e. the power is turned on. The transition between data retention and normal operation (active mode) is performed seamlessly without requiring any intervention from the host processor 106.

Referring next to FIG. 9, a block diagram is shown of a memory refresh system 900 using the selective refresh technique and the temperature compensated refresh techniques of FIGS. 7 and 8. Shown is a time of day clock 902, binary counters 904 and 906, comparators 908 and 910, partial refresh register 912, host processor 914 (host MCU 914), count register 916, memory array 918, digital temperature measurement 920, temperature sensor 922, refresh enable flip flop 924, clear signal 926, set signal 928, refresh enable signal 930 and refresh row enable 932.

Consistent with the descriptions associated with FIGS. 7 and 8, a digital temperature sensor 922 measures an ambient temperature. This temperature measurement 920 is sent to memory array 918 and translated into a count value according to a look up

table stored in the memory array, as described above with reference to FIG. 8. The count value corresponds to a time measurement in between refresh cycles for various operating temperatures. The count value is copied into a count register 916. This count value is then compared to the value of binary counter 906 by comparator 910 coupled therebetween. Binary counter 906 is coupled to the time of day clock 902 such that it counts according to real time.

When the value in binary counter 906 equals the count value in the count register 916, the comparator 910 outputs a set signal 928 to set the refresh enable flip-flop 924 to activate a refresh cycle, i.e. the refresh enable flip-flop 924 outputs a refresh enable signal 930 (e.g. refresh enable signal 930 goes high or "1"). At this point, after an amount of time determined according to the temperature the memory is refreshed. The set signal 928 is also output back to binary counter 906 (resetting it back to zero or another reference starting point) and to the digital temperature measurement 920, which is re-loaded into the memory array 918 in order to determine the next variable amount of the time for the next refresh cycle. If the temperature remains the same, the time in between refresh cycles remains the same. As the temperature changes, the time in between refresh cycles will change. For example, if the temperature drops, then the time in between refresh cycles is lengthened, advantageously conserving power since, the refresh cycle is less often.

Additionally, the host processor is coupled to a partial refresh register 912 such that the host processor 914 loads a value into the partial refresh register 912 indicating what portion of the system memory to refresh, e.g. which rows to refresh and which rows not to refresh. The value is the partial refresh register 912 is then compared to the value of binary counter 904 by comparator 908 coupled therebetween. Binary counter

904 is also coupled to the time of day clock 902, but does begin to start counting until comparator 910 outputs the set signal 928 which is also coupled to binary counter 904. Thus, once the refresh cycle (refresh enable signal is output) is activated by the set signal 928 into the refresh enable flip-flop 924, then the binary counter 904 begins to count.

Once the value in binary counter 904 equals the value in the partial refresh register 912, the comparator outputs a clear signal 926 to the refresh enable flip-flip 924 and back to the binary counter 904. The clear signal resets binary counter 904 and causes the refresh enable flip-flop to disable the refresh enable signal 930 (e.g. the refresh enable signal 930 goes low or "0"). This effectively stops the refresh process such that less than the entire portion of the memory is refreshed (as determined by the host processor prior to power off) which advantageously conserves power. Note that in this embodiment, the binary counter also counts a number corresponding to the refresh row address in memory (DRAM), i.e. refresh row address signal 932.

The memory refresh system shown in FIG. 9 advantageously uses both the selective refresh technique and the temperature compensated refresh technique to synergistically reduce power consumption during the refreshing of DRAM. The components used are well known in the art. Most of the components may be a part of the refresh controller 220 of FIG. 2 depending on the embodiment. For example, in one embodiment, the temperature sensor 922, temperature measurement 920 and memory array 918 are located on a separate chip, while the host processor is located on the same processor platform. The partial refresh register 912, comparators 908 and 910, binary counters 904 and 906, count register 916, and the refresh enable flip-flop 924 are all part of the refresh controller 220.

While the invention herein disclosed has been

described by means of specific embodiments and applications thereof, numerous modifications and variations could be made thereto by those skilled in the art without departing from the scope of the invention set forth in the claims.

## CLAIMS

What is claimed is:

1. A data transfer system comprising:

a plurality of peripheral interfaces;

a first memory; and

a programmable direct memory access module coupling the first memory to each of the plurality of peripheral interfaces, wherein the programmable direct memory access module configures selectively programmable direct memory access data channels between the first memory and respective ones of the plurality of peripheral interfaces.

2. The data transfer system of Claim 1 further comprising a first processor coupled to the programmable direct memory access module and associated with the first memory.

3. The data transfer system of Claim 1 further comprising a second memory coupled to the programmable direct memory access module, wherein the programmable direct memory access module configures the selectively programmable direct memory access data channels between the second memory and respective ones of the plurality of peripheral interfaces.

4. The data transfer system of Claim 3 further comprising a second processor coupled to the programmable direct memory access module and associated with the second memory.

5. The data transfer system of Claim 3 wherein the selectively programmable direct memory access data channels are configured between the first memory and the second memory.

6. The data transfer system of Claim 1 wherein the programmable direct memory access module includes a programmable processor.

7. The data transfer system of Claim 6 wherein the programmable direct memory access module further includes a direct memory access controller coupled to the programmable processor, wherein the programmable processor configures the selectively programmable direct memory access data channels between the first memory and respective ones of the plurality of peripheral interfaces via a dedicated direct memory access data transfer channel of the direct memory access controller.

8. The data transfer system of Claim 1 wherein the programmable direct memory access module further comprises a programmable scheduler for prioritizing data transfers over respective ones of the selectively programmable direct memory access data channels.

9. A direct memory access system comprising:

a direct memory access controller establishing a direct memory access data channel and including a first interface for coupling to a memory;

a second interface for coupling to a plurality of nodes; and

a processor coupled to the direct memory access controller and coupled to the second interface, wherein the processor configures the direct memory access data channel to transfer data between a programmably selectable respective one or more of the plurality of nodes and the memory.

10. The system of Claim 9 wherein the plurality of nodes are one of a plurality of peripheral interfaces and a memory interface.

11. The system of Claim 9 further comprising a programmable scheduler coupled to the processor for prioritizing the data transfer via the direct memory access data channel such that the data transfer occurs according to predetermined priorities.

12. A method for performing direct memory access, the method comprising:

receiving a request for a direct memory access data transfer;

configuring code to establish a direct memory access data transfer channel between a node specified by the request and a direct memory access interface; and

transferring data between the node and the direct memory access interface along the direct memory access data transfer channel.

13. The method of Claim 12 wherein the receiving step comprises receiving a timed request for a direct memory access data transfer.

14. The method of Claim 12 further comprising prioritizing the data to be transferred via the direct memory access data transfer channel.

15. The method of Claim 14 further comprising interrupting the transferring of data in the event a higher priority direct memory access data transfer is required.

16. The method of Claim 15 further comprising resuming the interrupted transfer of data upon completion of the higher priority direct memory access data transfer.

17. A memory access system comprising:

a memory;

a multiplexer coupled to the memory via a first bus having a first bus width;

a host processor coupled to the multiplexer via a second bus having a second bus width, wherein the second bus width is less than the first bus width; and

a display controller coupled to the multiplexer via a third bus having a third bus width the same as the first bus width to allow the memory to be utilized as a buffer for both the host processor and the display controller such that one of the display controller and the host processor may access the memory without restricting the ability of the other of the host processor or display controller to access the memory.

18. The memory access system of Claim 17 wherein the first bus width is 128 bits.

19. The memory access system of Claim 17 wherein the second bus width is a predetermined number of bits, the predetermined bit number being either 8, 16, or 32 bits.

20. The memory access system of Claim 17 further comprising a programmable direct memory access module coupled to the multiplexer via a fourth bus having a fourth bus width less than the first bus width to allow the memory to be used as a direct memory access node.

21. The memory access system of Claim 20 wherein the fourth bus width is a predetermined number of bits, the predetermined bit number being either 8, 16, or 32 bits.

22. The memory access system of Claim 17 wherein the memory is embedded onto a processor platform including the host processor and the display controller.

23. A method of memory refresh, the method comprising:

providing a dynamic random access memory having a predetermined capacity; and

periodically refreshing a portion of the dynamic random access memory less than its predetermined capacity to reduce power consumed as the memory portion is refreshed.

24. The method of Claim 23 further comprising adjusting a refresh rate of the dynamic random access memory based on ambient temperature, and memory is the periodically refreshed by periodically refreshing the portion at the adjusted refresh rate.

25. The method of Claim 23 wherein the memory capacity includes a predetermined total number of rows thereof and the memory portion comprises a number of rows of the dynamic random access memory less than its predetermined total number of rows.

26. A method of memory refresh, the method comprising:

providing a dynamic random access memory;

adjusting a refresh rate of the dynamic random access memory based on ambient temperature; and

periodically refreshing the dynamic random access memory at the adjusted refresh rate with refreshing occurring less frequently at lower ambient temperatures to reduce power consumed at lower temperatures.

27. The method of Claim 26 wherein the periodically refreshing step comprises periodically refreshing a portion of the dynamic random access memory less than its predetermined capacity of the memory to reduce power consumed as the memory portion is refreshed.

28. The method of Claim 26 wherein the memory capacity includes a predetermined total number of rows thereof and the memory portion comprises a number of rows of the dynamic random access memory less than its predetermined total number of rows.

29. A processor communication system, comprising:

a host processor direct memory access interface;

one or more peripheral ports;

a data bus for conveying data between said host processor direct memory access interface and said one or more peripheral ports; and

a programmable controller comprising a plurality of registers in communication with said data bus for maintaining data communication, with said programmable controller being operable for storing and retrieving data from the plurality of registers to establish multiple data transfers between said host processor direct memory access interface and said one or more peripheral ports.

30. A system as recited in claim 29, comprising a second processor direct memory access interface, wherein said programmable controller is operable for establishing multiple data transfers between said host processor direct memory access interface, said second processor direct memory access interface, and said one or more peripheral ports.

31. A system as recited in claim 30, wherein said programmable processor prioritizes the multiple data transfers established by storing and retrieving the data from the plurality of registers for data transfer between said host processor direct memory access interface, said second processor direct memory access interface, and said

one or more peripheral ports.

32. A system as recited in claim 30, wherein said second processor direct memory access interface comprises a digital signal processor direct memory access interface to random access memory associated with a digital signal processor.

33. A system as recited in claim 32, wherein said programmable controller is operable for establishing multiple virtual direct memory access channels between said host processor direct memory access interface, said digital signal processor, direct memory access interface, and said one or more peripheral ports.

34. A system as recited in claim 33, comprising embedded dynamic random access memory for use with the host processor.

35. A memory refresh system comprising:

a temperature sensor for providing a temperature measurement;

a memory coupled to the temperature sensor for providing a temperature dependent count value based upon the temperature measurement; and

a count register coupled to the memory for storing the temperature dependent count value used to determine a temperature dependent refresh cycle for a dynamic random access memory.

36. The system of Claim 35 further comprising:

a comparator coupled to the count register; and

a counter coupled to the comparator, wherein the comparator outputs a signal when the counter matches the temperature dependent count value within the count register.

37. The system of Claim 36 further comprising a refresh enable unit coupled to the comparator, wherein the refresh enable unit provides a refresh enable signal responsive to the signal output from the comparator, wherein providing the temperature dependent refresh cycle for the dynamic random access memory.

## ABSTRACT OF THE DISCLOSURE

A direct memory access system consists of a direct memory access controller establishing a direct memory access data channel and including a first interface for coupling to a memory. A second interface is for coupling to a plurality of nodes. And a processor is coupled to the direct memory access controller and coupled to the second interface, wherein the processor configures the direct memory access data channel to transfer data between a programmably selectable respective one or more of the plurality of nodes and the memory. In some embodiments, the plurality of nodes are a digital signal processor memory and a host processor memory of a multi-media processor platform to be implemented in a wireless multi-media handheld telephone.

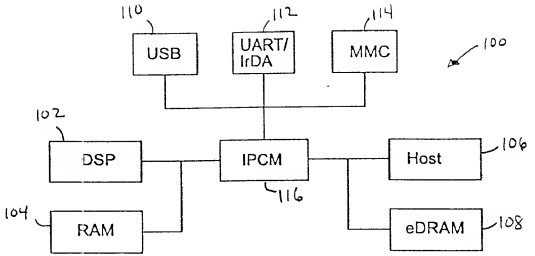

110
USB
112
UART/
IrDA
114
MMC
100
102
DSP
IPCM
Host
106
116
104
RAM
eDRAM
108

FIG. 1

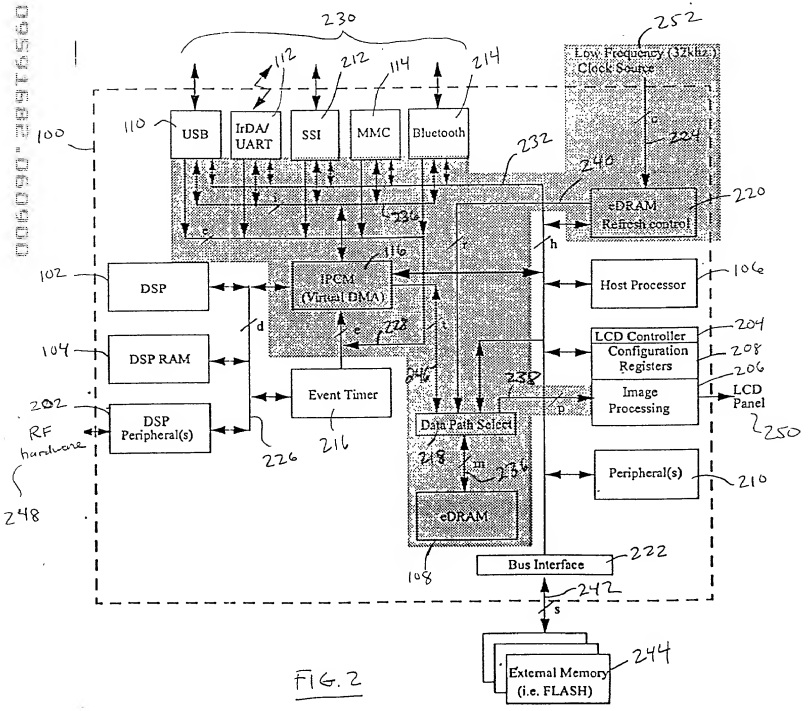

230
252
Low Frequency (32khz) Clock Source
112
212
114
214
110
USB
IrDA/ UART
SSI
MMC
Bluetooth
100
232
240
224
236
eDRAM
Refresh control
220
102
DSP
116
IPCM
(Virtual DMA)
Host Processor
106
104
DSP RAM
222
LCD Controller
204
Configuration Registers
208
206
238
Image Processing
LCD Panel
250
202
RF hardware
DSP Peripheral(s)
Event Timer
Data Path Select
218
234
Peripheral(s)
210
226
216
248
eDRAM
108
Bus Interface
222
242
244
External Memory (i.e. FLASH)

FIG. 2

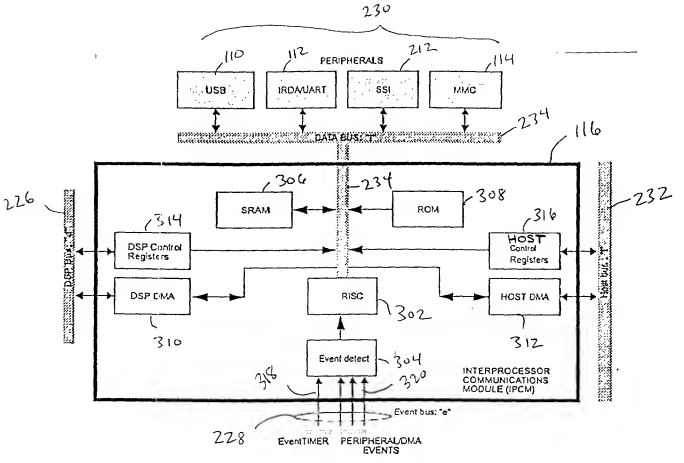

230
110
112
PERIPHERALS
212
114
USB
IRDA/UART
SSI
MMC
234
DATA BUS
116
226
306
234
SRAM
ROM
308
232
314
316
DSP Control Registers
HOST Control Registers
DSP DMA
RISC
HOST DMA
302
310
312
Event detect
304
318
320
INTERPROCESSOR COMMUNICATIONS MODULE (IPCM)
Event bus: "e"
228
EventTIMER
PERIPHERAL/DMA EVENTS

FIG. 3

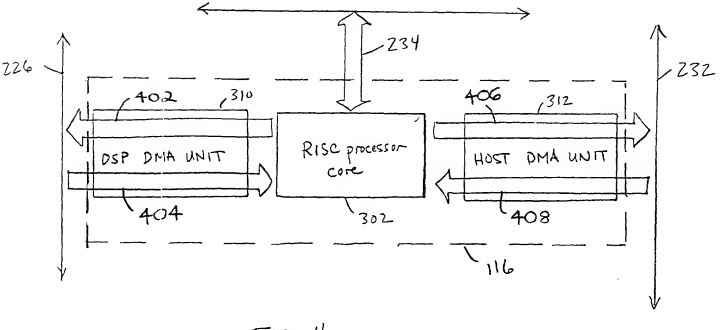

234
226
232
402
310
406
312
DSP DMA UNIT
RISC processor core
HOST DMA UNIT
404
302
408
116

FIG. 4

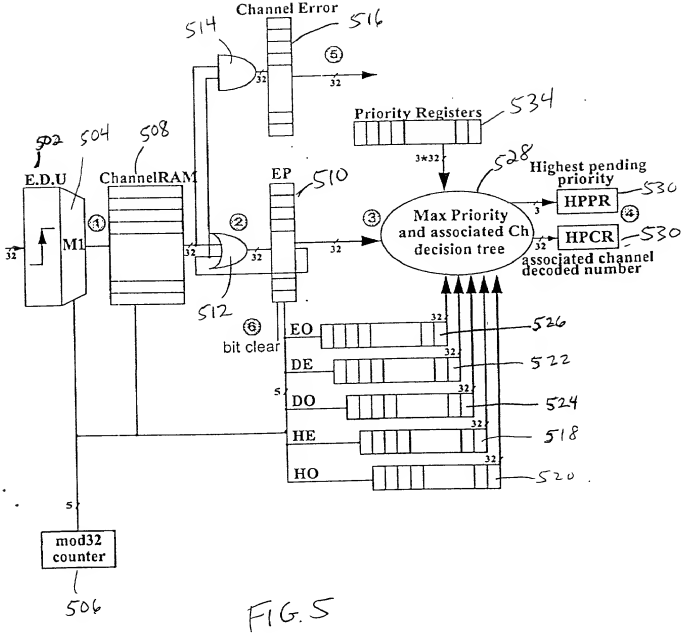

Channel Error
514
516
5
32
Priority Registers
534
3*32
528
Highest pending priority
HPPR
530
3
4
HPCR
530
32
associated channel decoded number
Max Priority and associated Ch decision tree
502
504
508
E.D.U
ChannelRAM
EP
510
M1
1
2
3
32
512
6
bit clear
EO
526
DE
522
DO
524
HE
518
HO
520
5
mod32 counter
506

FIG. 5

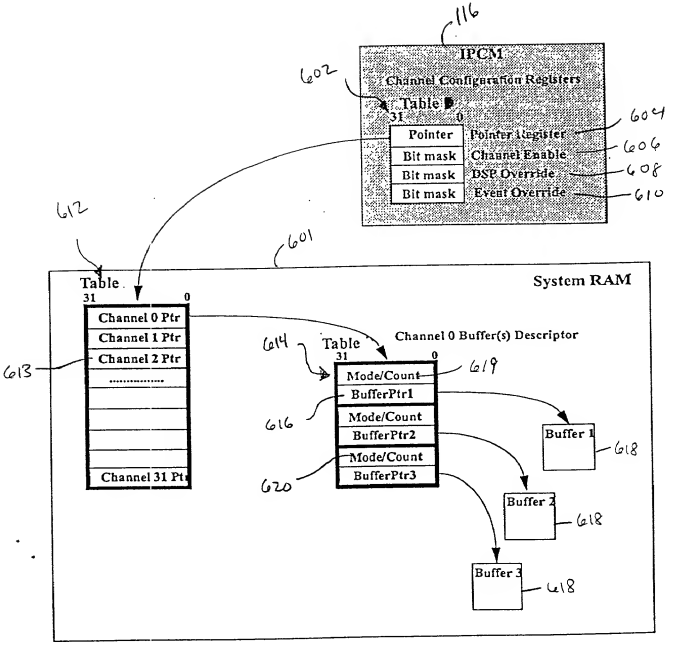

116
IPCM
Channel Configuration Registers
602
Table
31
0
Pointer
Pointer Register
604
Bit mask
Channel Enable
606
Bit mask
DSP Override
608
Bit mask
Event Override
610
612
601
Table
31
0
Channel 0 Ptr
Channel 1 Ptr
Channel 2 Ptr
Channel 31 Ptr
613
System RAM
614
Table
Channel 0 Buffer(s) Descriptor
31
0
Mode/Count
619
BufferPtr1
616
Mode/Count
BufferPtr2
Mode/Count
620
BufferPtr3
Buffer 1
618
Buffer 2
618
Buffer 3
618

FIG. 6

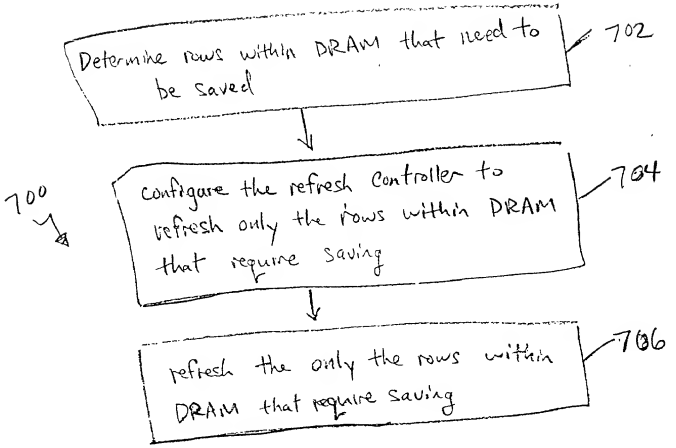

Determine rows within DRAM that need to be saved
702
700
configure the refresh controller to refresh only the rows within DRAM that require saving
704
refresh the only the rows within DRAM that require saving
706

FIG. 7

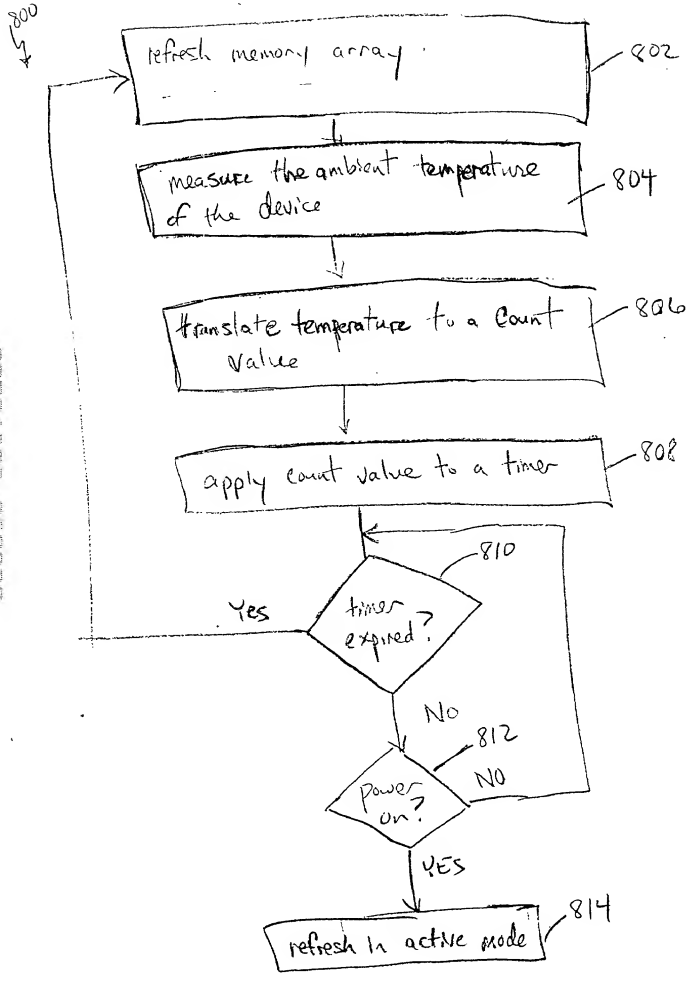

800
refresh memory array
802
measure the ambient temperature of the device
804
translate temperature to a count value
806
apply count value to a timer
808
timer expired?
810
Yes
No
power on?
812
NO
YES
refresh in active mode
814

FIG. 8

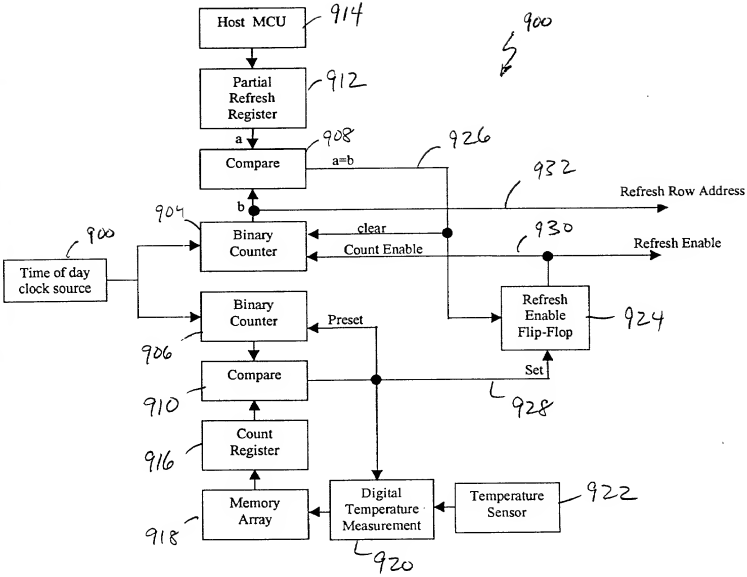

Host MCU
914
900
Partial Refresh Register
912
a
908
926
Compare
a=b
932
Refresh Row Address
b
904
clear
930
Binary Counter
Count Enable
Refresh Enable
900
Time of day clock source
Binary Counter
Preset
Refresh Enable Flip-Flop
924
906
Compare
Set
910
928
Count Register
916
Memory Array
Digital Temperature Measurement
Temperature Sensor
922
918
920

FIG. 9

**APPENDIX B**

EXAMPLE ROM SCRIPTS FOR RISC
PROCESSOR CORE OF IPCM

## 9.13 Example Scripts (ROM contents)

### 9.13.1 Vectors

The first 32 addresses are jumps to routines (reset at address 0, illegal instruction at address 1, etc.)

```
0     8051  // _start:   jmp GetCh0PC
1     8001  // 1:        jmp 1b          // 1 - illegal instruction trap
2     80fc  //           jmp MDdma       // 2 - copy from MCore to DSP
3     8150  //           jmp DMdma       // 3 - copy from DSP to MCore
4     0088  //           NOP (mov r0,r0)
5     0088  //           NOP
6     0088  //           NOP
7     0088  //           NOP
8     0088  //           NOP
9     0088  //           NOP
10    0088  //           NOP
11    0088  //           NOP
12    0088  //           NOP
13    0088  //           NOP
14    0088  //           NOP
15    0088  //           NOP
16    0088  //           NOP
17    0088  //           NOP
18    0088  //           NOP
19    0088  //           NOP
```

```
20    0088  //          NOP
21    0088  //          NOP
22    0088  //          NOP
23    0088  //          NOP
24    0088  //          NOP
25    0088  //          NOP
26    0088  //          NOP
27    0088  //          NOP
28    0088  //          NOP
29    0088  //          NOP
30    0088  //          NOP
31    0088  //          NOP
```

### 9.13.2 Context Switch

The following code is the context switch routine in ROM; notice that almost all the instructions here cannot work anywhere else; the execution order is the only possible one.

```
32    00e3  //          CtxPtrInit        // DM[0x7002] --> r0
33    01e3  //          CatchSPtr         // address to spill registers
34    00e0  //          ldMAstGP          // (GReg[1] saved) & (MA --> GReg[1])
35    01e0  //          ldMDstG2          // (GReg[2] saved) & (MD --> GReg[2])
36    02e0  //          ldMSstG3          // (GReg[3] saved) & (MS --> GReg[3])
37    03e0  //          ldDAstG4          // (GReg[4] saved) & (DA --> GReg[4])
38    04e0  //          ldDDstG5          // (GReg[5] saved) & (DD --> GReg[5])
39    05e0  //          ldDSstG6          // (GReg[6] saved) & (DS --> GReg[6])
40    02e3  //          stG7mvShPC        // (GReg[7] s.) & (ShPC --> GReg[7])
41    03e3  //          stG7mvShLoop      // (ShPC saved) & (ShLoop --> GReg[7])
42    07e0  //          stG7ldCS          // (ShLoop saved) & (CS --> GReg[7])
43    00e0  //          ldMAstG1          // (MA saved)
44    01e0  //          ldMDstG2          // (MD saved)
45    02e0  //          ldMSstG3          // (MS saved)
46    03e0  //          ldDAstG4          // (DA saved)
47    04e0  //          ldDDstG5          // (DD saved)
48    05e0  //          ldDSstG6          // (DS saved)
49    06e0  //          ldCAstG7          // (CS saved)
50    01e2  //          stCAmovShReg02Gr1
51    00e3  //          CtxPtrInit
52    01e3  //          CatchCPtr
53    01e4  //          TstPendingAndSwitch// ShReg0 saved
54    00e3  //          CtxPtrInit        // DM[0x7002] --> GReg[0]
55    01e3  //          CatchCPtr         // Get address for restoring registers
56    1811  //          addi r0,#17       // Points to CA of channel to restore
57    04e3  //          ldFU0inLd0
58    05e3  //          mvFU02G1          // GReg[1] = CA
59    07e1  //          ldmfub7           // CA restored & fetch CS
60    06e1  //          ldmfub6           // CS restored & fetch DS
61    05e1  //          ldmfub5           // DS restored & fetch DD
62    04e1  //          ldmfub4           // DD restored & fetch DA
63    03e1  //          ldmfub3           // DA restored & fetch MS
64    02e1  //          ldmfub2           // MS restored & fetch MD
65    01e1  //          ldmfub1           // MD restored & fetch MA
```

```
66    00e1  //          ldmfub0          // MA restored & fetch ShLoop
67    06e3  //          ldShLoop         // ShLoop restored & fetch ShPC
68    07e3  //          ldShPC           // ShPC restored & fetch GReg[7]
69    07e2  //          ldmGReg7         // GReg[7]restored & fetch GReg[6]
70    06e2  //          ldmGReg6         // GReg[6]restored & fetch GReg[5]
71    05e2  //          ldmGReg5         // GReg[5]restored & fetch GReg[4]
72    04e2  //          ldmGReg4         // GReg[4]restored & fetch GReg[3]
73    03e2  //          ldmGReg3         // GReg[3]restored & fetch GReg[2]
74    02e2  //          ldmGReg2         // GReg[2]restored & fetch GReg[1]
75    00e2  //          ldmGReg1GReg0    // GReg[1] & GReg[0] restored
76    00e4  //          cpShReg          // copy all shadow registers (PC, ...)
77    0088  //          NOP
78    0088  //          NOP
79    0088  //          NOP
```

### 9.13.3 Boot Code

The standard channel 0 boot code.

```
80    80ba  //          jmp Reset        // jump around common subroutines
81    0870  // GetCh0PC:ldi r0,REGPAGE
82    0011  //          revblo r0        // r0 = 0x7000
83    51e0  //          ld rT,(r0,CHN0ADDR)// load first pc value of channel 0
84    0108  //          jmpr rT
85    0300  // 0:       done 3           //clear HE set HI
86    0a70  // Mccb:    ldi rC,REGPAGE
87    0211  //          revblo rC        // pointer to register page
88    501a  //          ld r0,(rC,R_CCR) // load current channel number
89    5202  //          ld rC,(rC,R_MC0PTR)// load channel 0 ptr
90    4a00  //          cmpeqi rC,0
91    7df9  //          bt 0b
92    0017  //          lsll r0
93    0017  //          lsll r0
94    0017  //          lsll r0
95    0017  //          lsll r0
96    0298  //          add rC,r0
97    0006  // 0:       ret
98    0700  // 0:       done 7           // clear DE set DI
99    0a70  // Dccb:    ldi rC,REGPAGE
100   0211  //          revblo rC        // pointer to register page
101   501a  //          ld r0,(rC,R_CCR) // load current channel number
102   520a  //          ld rC,(rC,R_DC0PTR)// load channel 0 ptr
103   4a00  //          cmpeqi rC,0
104   7df9  //          bt 0b
105   0017  //          lsll r0
106   0017  //          lsll r0
107   0017  //          lsll r0
108   0017  //          lsll r0
109   0298  //          add rC,r0
110   0006  // 0:       ret
111   0007  // MDone:   clrf 0           // clear flags
112   6b00  //          stf rP,ma        // address of status word
```

```
113  3501  //           andni rS,BD_DONE // indicate complete
114  6d12  //           stf rS,md,sz16   // update BD flags (DO NOT FLUSH!)
115  6e16  //           stf rN,md,sz16,fls// update BD count
116  7f18  //           bdf Mcerr
117  4508  //           tsti rS,BD_INTR // should we interrupt?
118  7c01  //           bf 0f
119  0201  //           notify 2        // yes, do it
120  1b08  // 0:        addi rP,BD_SIZE // bump ptr, assume no wrap
121  4502  //           tsti rS,BD_WRAP // are we really at the end
122  7c06  //           bf 1f           // if not, skip the following
123  0007  // MFirst:   clrf 0          // clr flags, could enter here
124  038a  //           mov rP,rC
125  1b04  //           addi rP,CB_BPTR // offset to ptr to BD array
126  6b04  //           stf rP,ma,pre
127  6313  //           ldf rP,md,sz32  // get BD array pointer
128  7e0c  //           bsf Mcerr
129  0006  // 1:        ret
130  0300  // Mblock:   done 3          // must wait on MCore clear HE set HI
131  0007  // MGetBD:   clrf 0          // make sure flags are clear
132  6b04  //           stf rP,ma,pre   // new DMA address to load from
133  6516  //           ldf rS,md,sz16,pre// get status from buf desc
134  7e06  //           bsf Mcerr
135  4501  //           tsti rS,BD_DONE // check for done bit
136  7cf9  //           bf Mblock       // if zero, not a valid BD
137  6616  //           ldf rN,md,sz16,pre// get byte count from BD
138  6413  //           ldf rA,md,sz32  // get address from BD
139  7e01  //           bsf Mcerr
140  0006  //           ret
141  808d  // Mcerr:9: jmp 9b
142  0300  // MblockCh0:done 3          // must wait on MCore clear HE set HI
143  8051  //           jmp GetCh0PC    // specific for Channel 0
144  0007  // MGetBDCh0:clrf 0          // make sure flags are clear
145  6b04  //           stf rP,ma,pre   // new DMA address to load from
146  6516  //           ldf rS,md,sz16,pre// get status from buf desc
147  7e06  //           bsf McerrCh0
148  4501  //           tsti rS,BD_DONE // check for done bit
149  7cf8  //           bf MblockCh0    // if zero, not a valid BD
150  6616  //           ldf rN,md,sz16,pre// get byte count from BD
151  6413  //           ldf rA,md,sz32  // get address from BD
152  7e01  //           bsf McerrCh0
153  0006  //           ret
154  809a  // McerrCh0:9:jmp 9b
155  0007  // DDone:    clrf 0          // clear flags
156  6b20  //           stf rP,da       // address of status word
157  3501  //           andni rS,BD_DONE// indicate complete
158  6d32  //           stf rS,dd,sz16  // update BD flags  (DO NOT FLUSH!)
159  6e36  //           stf rN,dd,sz16,fls// update count
160  7f18  //           bdf Dcerr
161  4508  //           tsti rS,BD_INTR // should we interrupt?
162  7c01  //           bf 0f
163  0601  //           notify 6        // yes, do it
164  1b08  // 0:        addi rP,BD_SIZE // bump ptr, assume no wrap
165  4502  //           tsti rS,BD_WRAP // are we really at the end
166  7c06  //           bf 1f           // if not, skip the following
```

```
167  0007  // DFirst:  clrf 0          // clr flags, could enter here
168  038a  //          mov rP,rC
169  1b04  //          addi rP,CB_BPTR // offset to ptr to BD array
170  6b24  //          stf rP,da,pre
171  6333  //          ldf rP,dd,sz32  // get BD array pointer
172  7e0c  //          bsf Dcerr
173  0006  // 1:       ret
174  0700  // Dblock:  done 7          // must wait on DSP - clear DE set DI
175  0007  // DGetBD:  clrf 0          // make sure flags are clear
176  6b24  // 0:       stf rP,da,pre   // new DMA address to load from
177  6536  //          ldf rS,dd,sz16,pre// get status from buf desc
178  7e06  //          bsf Dcerr
179  4501  //          tsti rS,BD_DONE // check for done bit
180  7cf9  //          bf Dblock       // if zero, not a valid BD
181  6636  //          ldf rN,dd,sz16,pre// get byte count from buf desc
182  6433  //          ldf rA,dd,sz32  // get address from buf desc
183  7e01  //          bsf Dcerr
184  0006  //          ret
185  80b9  // Dcerr:9: jmp 9b
186  0970  // Reset:   ldi rT,REGPAGE
187  0111  //          revblo rT       // pointer to register page
188  5201  //          ld rC,(rT,R_MCOPTR)// load channel 0 ptr
189  4a00  //          cmpeqi rC,0     // is channel 0 pointer zero
190  7c03  //          bf 0f           // no, continue
191  0100  //          done 1          // yes, shut down channel - reschedule
192  8051  //          jmp GetCh0PC    // Get the first PC value of channel 0
193  80ba  //          jmp Reset       // then start over
194  c07b  // 0:       jsr MFirst      // get address of first BD
195  0f00  //          ldi rL,0        // in case forget to set addr
196  80c6  //          jmp 3f
197  c06f  // 2:       jsr MDone
198  c090  // 3:       jsr MGetBDCh0
199  018d  //          mov rT,rS
200  0112  //          rorb rT         // get command in low byte
201  3907  //          andi rT,0x07    // look at 3 low bits
202  4901  //          cmpeqi rT,C0_ADDR// command = set address?
203  7c02  //          bf 1f           // no, try next command code
204  078c  //          mov rL,rA       // save address field
205  80c5  //          jmp 2b
206  4902  // 1:       cmpeqi rT,C0_LOAD// load command
207  7c0c  //          bf 1f           // no, try next command code
208  6c04  // 4:       stf rA,ma,pre
209  008e  //          mov r0,rN
210  0015  //          lsr1 r0         // byte count -> word count
211  0015  //          lsr1 r0
212  7804  //          loop 4,0
213  6117  //          ldf rT,md,sz32,pre
214  5907  //          st rT,(rL,0)
215  1f01  //          addi rL,1
216  0000  //          yield
217  7e17  //          bsf c0berr
218  7c01  //          bf 1f
219  80c5  //          jmp 2b
220  4903  // 1:       cmpeqi rT,C0_DUMP// dump command
```

```
221  7c0b  //          bf 1f             // no, try next command code
222  6c00  // 5:       stf rA,ma
223  008e  //          mov r0,rN
224  0015  //          lsr1 r0           // byte count -> word count
225  0015  //          lsr1 r0
226  7804  //          loop 4,0
227  5107  //          ld rT,(rL,0)
228  6917  //          stf rT,md,sz32,fls
229  1f01  //          addi rL,1
230  0000  //          yield
231  7e09  //          bsf c0berr
232  80c5  //          jmp 2b
233  4904  // 1:       cmpeqi rT,C0_SETCTX// set context command
234  7c02  //          bf 1f             // no, try next command code
235  c0f2  //          jsr CmdCtx
236  80d0  //          jmp 4b
237  4905  // 1:       cmpeqi rT,C0_GETCTX// get context command
238  7cca  //          bf 9b             // no, ignore
239  c0f2  //          jsr CmdCtx
240  80de  //          jmp 5b
241  80f1  // c0berr:9:jmp 9b
242  0f08  // CmdCtx:  ldi rL,CTXPAGE
243  0711  //          revblo rL         // address of contexts
244  018d  //          mov rT,rS
245  0112  //          rorb rT
246  39f8  //          andi rT,0xF8      // keep 5 bits (chan * 8)
247  0799  //          add rL,rT         // base += (chan * 8)
248  0799  //          add rL,rT         // base += (chan * 16)
249  0115  //          lsr1 rT           // (chan * 4)
250  0799  //          add rL,rT         // base += (chan * 20)
251  0006  //          ret
```

### 9.13.4 MCU to DSP transfer

```
252  c056  // MDdma:   jsr Mccb          // get MCore CCB address
253  5a07  //          st rC,(rL,LS_MC) // save it
254  c07b  //          jsr MFirst        // get MCore first BD address
255  5b0f  //          st rP,(rL,LS_MP)
256  c063  //          jsr Dccb          // get DSP CCB address
257  5a27  //          st rC,(rL,LS_DC) // save it
258  c0a7  //          jsr DFirst        // get DSP first BD address
259  5b2f  //          st rP,(rL,LS_DP)
260  0a00  //          ldi rC,0
261  530f  // MDMnxt:  ld rP,(rL,LS_MP) // enter with rC = DSP resid
262  c083  //          jsr MGetBD        // get next MCore BD
263  5b0f  //          st rP,(rL,LS_MP)
264  5d17  //          st rS,(rL,LS_MS)
265  5e1f  //          st rN,(rL,LS_MN)
266  6c04  //          stf rA,ma,pre     // load MCore DMA address
267  008e  //          mov r0,rN         // save count
268  4a00  //          cmpeqi rC,0
269  7c08  //          bf 2f
270  532f  // MDDnxt:  ld rP,(rL,LS_DP) // enter with r0 = MCore resit
```

```
271  c0af  //          jsr DGetBD       // get next DSP BD
272  5b2f  //          st rP,(rL,LS_DP)
273  5d37  //          st rS,(rL,LS_DS)
274  5e3f  //          st rN,(rL,LS_DN)
275  6c20  //          stf rA,da        // load DSP DMA address
276  028e  //          mov rC,rN        // put DSP count in rC
277  0688  //          mov rN,r0        // put MCore count in rN
278  00d2  // 2:       cmplt r0,rC      // calculate loop count
279  7d01  //          bt 3f
280  008a  //          mov r0,rC        // DSP count < MCore count
281  06a0  // 3:       sub rN,r0        // adjust MCore count
282  02a0  //          sub rC,r0        // adjust DSP count
283  4800  //          cmpeqi r0,0      // is trip count 0?
284  7d08  //          bt 4f
285  1803  //          addi r0,3        // round up from byte count
286  0015  //          lsr1 r0          //
287  0015  //          lsr1 r0          // to full word count
288  7803  //          loop 3,0         // clr flags, loop on next 3
289  6117  //          ldf rT,md,sz32,pre// LOOP: get 32 bits from MCore
290  6933  //          stf rT,dd,sz32   // LOOP: put 32 bits to DSP
291  0000  //          done 0           // LOOP: yield to > priority
292  7c2a  //          bf MDerr         // branch if not count runout
293  4e00  // 4:       cmpeqi rN,0      // MCore count reach zero?
294  7c16  //          bf MDMnz         // branch if nonzero
295  008a  //          mov r0,rC        // put DSP resid count in r0
296  5207  //          ld rC,(rL,LS_MC) // close MCore BD
297  530f  //          ld rP,(rL,LS_MP)
298  5517  //          ld rS,(rL,LS_MS)
299  561f  //          ld rN,(rL,LS_MN)
300  c06f  //          jsr MDone
301  5b0f  //          st rP,(rL,LS_MP)
302  563f  //          ld rN,(rL,LS_DN)
303  4800  //          cmpeqi r0,0      // DSP count reach zero?
304  7d03  //          bt 0f            // branch if zero
305  5117  //          ld rT,(rL,LS_MS)
306  4104  //          tsti rT,BD_CONT  // is MCore continue bit set?
307  7d07  //          bt 1f            // yes, don't close DSP BD
308  06a0  // 0:       sub rN,r0        // close DSP BD
309  5227  //          ld rC,(rL,LS_DC)
310  532f  //          ld rP,(rL,LS_DP)
311  5537  //          ld rS,(rL,LS_DS)
312  c09b  //          jsr DDone
313  5b2f  //          st rP,(rL,LS_DP)
314  0800  //          ldi r0,0
315  0288  // 1:       mov rC,r0        // put DSP resid back in rC
316  8105  //          jmp MDMnxt
317  008e  // MDMnz:   mov r0,rN        // put MCore resid count in r0
318  5227  //          ld rC,(rL,LS_DC) // close DSP BD
319  532f  //          ld rP,(rL,LS_DP)
320  5537  //          ld rS,(rL,LS_DS)
321  563f  //          ld rN,(rL,LS_DN)
322  c09b  //          jsr DDone
323  5b2f  //          st rP,(rL,LS_DP)
324  4504  //          tsti rS,BD_CONT  // is DSP continue bit set?
```

```
325   7dc8  //            bt MDDnxt
326   561f  //            ld rN,(rL,LS_MN) // close MCore BD
327   06a0  //            sub rN,r0
328   5207  //            ld rC,(rL,LS_MC)
329   530f  //            ld rP,(rL,LS_MP)
330   5517  //            ld rS,(rL,LS_MS)
331   c06f  //            jsr MDone
332   5b0f  //            st rP,(rL,LS_MP)
333   0a00  //            ldi rC,0
334   8105  //            jmp MDMnxt
335   814f  // MDerr:9:   jmp 9b
```

### 9.13.5 DSP to MCU transfer

```
336   c063  // DMdma:    jsr Dccb          // get DSP CCB address
337   5a27  //            st rC,(rL,LS_DC) // save it
338   c0a7  //            jsr DFirst        // get DSP first BD address
339   5b2f  //            st rP,(rL,LS_DP)
340   c056  //            jsr Mccb          // get MCore CCB address
341   5a07  //            st rC,(rL,LS_MC) // save it
342   c07b  //            jsr MFirst        // get MCore first BD address
343   5b0f  //            st rP,(rL,LS_MP)
344   0a00  //            ldi rC,0
345   532f  // DMDnxt:    ld rP,(rL,LS_DP)
346   c0af  //            jsr DGetBD        // get next DSP BD
347   5b2f  //            st rP,(rL,LS_DP)
348   5d37  //            st rS,(rL,LS_DS)
349   5e3f  //            st rN,(rL,LS_DN)
350   6c24  //            stf rA,da,pre     // load DSP DMA addr, prefetch
351   008e  //            mov r0,rN         // save count
352   4a00  //            cmpeqi rC,0
353   7c08  //            bf 2f
354   530f  // DMMnxt:    ld rP,(rL,LS_MP)
355   c083  //            jsr MGetBD        // get next MCore BD
356   5b0f  //            st rP,(rL,LS_MP)
357   5d17  //            st rS,(rL,LS_MS)
358   5e1f  //            st rN,(rL,LS_MN)
359   6c00  //            stf rA,ma         // load MCore DMA address
360   028e  //            mov rC,rN         // put MCore count in rC
361   0688  //            mov rN,r0         // put DSP count in rN
362   00d2  // 2:         cmplt r0,rC       // calculate loop count
363   7d01  //            bt 3f
364   008a  //            mov r0,rC
365   06a0  // 3:         sub rN,r0         // adjust DSP count
366   02a0  //            sub rC,r0         // adjust MCore count
367   4800  //            cmpeqi r0,0       // is trip count 0?
368   7d08  //            bt 4f
369   1803  //            addi r0,3         // round up from byte count
370   0015  //            lsrl r0
371   0015  //            lsrl r0           // to full word count
372   7803  //            loop 3,0          // clr flags, loop on next 3
373   6137  //            ldf rT,dd,sz32,pre// LOOP: get 32 bits from DSP
374   6913  //            stf rT,md,sz32    // LOOP: put 32 bits to MCore
```

```
0000  //          done 0          // LOOP: yield to > priority
7c2a  //          bf DMerr        // branch if not count runout
4e00  // 4:       cmpeqi rN,0     // DSP count reach zero?
7c16  //          bf DMDnz        // branch if nonzero
008a  //          mov r0,rC       // put MCore resid count in r0
5227  //          ld rC,(rL,LS_DC) // close DSP BD
532f  //          ld rP,(rL,LS_DP)
5537  //          ld rS,(rL,LS_DS)
563f  //          ld rN,(rL,LS_DN)
c09b  //          jsr DDone
5b2f  //          st rP,(rL,LS_DP)
561f  //          ld rN,(rL,LS_MN)
4800  //          cmpeqi r0,0     // MCore count reach zero?
7d03  //          bt 0f           // branch if zero
5137  //          ld rT,(rL,LS_DS)
4104  //          tsti rT,BD_CONT // is DSP continue bit set
7d07  //          bt 1f           // yes, don't close MCore BD
06a0  // 0:       sub rN,r0       // close MCore BD
5207  //          ld rC,(rL,LS_MC)
530f  //          ld rP,(rL,LS_MP)
5517  //          ld rS,(rL,LS_MS)
c06f  //          jsr MDone
5b0f  //          st rP,(rL,LS_MP)
0800  //          ldi r0,0
0288  // 1:       mov rC,r0       // put MCore resid back in rC
8159  //          jmp DMDnxt
008e  // DMDnz:   mov r0,rN       // put DSP resid count in r0
5207  //          ld rC,(rL,LS_MC) //close MCore BD
530f  //          ld rP,(rL,LS_MP)
5517  //          ld rS,(rL,LS_MS)
561f  //          ld rN,(rL,LS_MN)
c06f  //          jsr MDone
5b0f  //          st rP,(rL,LS_MP)
4504  //          tst irS,BD_CONT // is MCore continue bit set?
7dc8  //          bt DMNxt
563f  //          ld rN,(rL,LS_DN) //close DSP BD
06a0  //          sub rN,r0
5227  //          ld rC,(rL,LS_DC)
532f  //          ld rP,(rL,LS_DP)
5537  //          ld rS,(rL,LS_DS)
c09b  //          jsr DDone
5b2f  //          st rP,(rL,LS_DP)
0a00  //          ldi rC,0
8159  //          jmp DMDnxt
81a3  // DMerr:9: jmp 9b
```

### 9.13.6 PPP routine

```
0802  // ppprinit:ldi r0,0x2
6840  //          stf r0,ca       // select CRC-CCITT16
c056  //          jsr Mccb        // get MCore CCB address
c07b  //          jsr MFirst      // get BD array pointer of channel
c083  // pppmxt:  jsr MGetBD      // get first Buffer Decriptor
```

```
425  6c00  //           stf rA,ma          // buffer address
426  008e  //           mov r0,rN          // buffer size
427  0900  //           ldi rT,0
428  2101  //           subi rT,1          // RT=0xFFFFFFFF
429  6948  //           stf rT,cs,0        // initialize CRC
430  81b0  //           jmp 2f             // To avoid the wait instruction
431  0400  // 1:        done 4
432  511f  // 2:        ld rT,(rL,RECV)
433  7efd  //           bsf 1b             // no char, or framing error
434  497e  //           cmpeqi rT,0x7E
435  7cfc  //           bf 2b              // not a flag char
436  7807  // ppprlp:   loop 7,0
437  511f  // 4:        ld rT,(rL,RECV)
438  497d  //           cmpeqi rT,0x7D     // check for ESC
439  7d06  //           bt 7f              // it is an ESC
440  497e  //           cmpeqi rT,0x7E     // check for ending flag
441  7d0d  //           bt 9f              // it is the ending flag
442  6949  // 5:        stf rT,cs,run      // run CRC on byte
443  6911  //           stf rT,md,sz8      // one byte DMA write
444  7f04  //           bdf 8f             // check for DMA error
445  81c3  //           jmp HOP
446  511f  // 7:        ld rT,(rL,RECV)    // get the data after the ESC
447  1120  //           xori rT,0x20       // de-escape the byte
448  81ba  //           jmp 5b             // go store it
449  2d10  // 8:        ori rS,BD_MERR
450  81ce  //           jmp 1f
451  7e01  // HOP:      bsf UART_WKr       // UART watermark has been reached.
452  81c7  //           jmp 0f             // Means loop ended with no error
453  0400  // UART_WKr:done 4
454  81b4  //           jmp ppprlp
455  6148  // 0:9:      ldf rT,cs
456  0ef0  //           ldi rN,0xf0
457  0611  //           revblo rN
458  1eb8  //           addi rN,0xB8       // desired final crc value 0xf0b8
459  01ce  //           cmpeq rT,rN        // compare for equal
460  7d01  //           bt 1f
461  2d40  //           ori rS,BD_CRCE
462  0688  // 1:        mov rN,r0          // size of data transferred
463  c06f  //           jsr MDone          // finish BD
464  81a8  //           jmp pppnxt
465  0802  // pppxinit:ldi r0,0x2
466  6840  //           stf r0,ca          // select CRC-CCITT16
467  c055  //           jsr Mccb           // get MCore CCB address
468  c07b  //           jsr MFirst
469  81d7  //           jmp pppxnxt
470  c06f  // pppxdon:  jsr MDone
471  c083  // pppxnxt:  jsr MGetBD         // don't touch rN,rS,rL
472  6c04  //           stf rA,ma,pre      // buffer address
473  008e  //           mov r0,rN          // buffer size
474  0900  //           ldi rT,0
475  2101  //           subi rT,1          // RT=0xFFFFFFFF
476  6948  //           stf rT,cs,0        // initialize CRC
477  0c7e  //           ldi rA,0x7E        // flag
478  5c27  //           st rA,(rL,XMIT)    // opening flag
```

```
479   780a  // pppxlp:   loop 10,0
480   6115  //          ldf rT,md,sz8,pre// get next data byte
481   497e  //          cmpeqi rT,0x7E  // need to escape escape?
482   7d03  //          bt 1f
483   497d  //          cmpeqi rT,0x7D  // need to escape escape?
484   7d01  //          bt 1f
485   7c03  //          bf 3f
486   1120  // 1:       xori rT,0x20    // modify escaped data (rT = 5E or 5D)
487   0c7d  //          ldi rA,0x7D     // escape
488   5c27  //          st rA,(rL,XMIT) // send escape character
489   5927  // 3:       st rT,(rL,XMIT) // send data
490   7f01  //          bdf UART_WKx    // UART watermark has been reached
491   81ee  //          jmp 0f          // loop ended with no error
492   0400  // UART_WKx:done 4          // awake an other channel
493   81df  //          jmp pppxlp      // lm flag is lost when loop is exited
494   6148  // 0:       ldf rT,cs       // get the 1021 data
495   0111  //          revblo rT
496   5927  //          st rT,(rL,XMIT) // crc hi byte
497   0111  //          revblo rT
498   5927  //          st rT,(rL,XMIT) // crc lo byte
499   0c7e  //          ldi rA,0x7E
500   5c27  //          st rA,(rL,XMIT) // ending flag
501   81d6  //          jmp pppxdon
```

**APPENDIX A**

INSTRUCTION SET FOR RISC
PROCESSOR CORE OF IPCM

# ADD Addition

**Operation:**

```
GReg[r] ← GReg[s] + GReg[r]
T ← (GReg[r] == 0)
```

**Assembler**
Syntax: `add r,s`

Example: `add 0,3`
to ADD GReg[3] and GReg[0] and store the result in GReg[0]

CPU Flags: T

Cycles: 1

Description: Performs the ADDition of the source General Register s and the destination General Register r, and stores the result in the destination General Register r. The T flag is set if the result of the operation is 0; it is cleared if the result is not zero.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | r | r | r | 1 | 0 | 0 | 1 | 1 | s | s | s |

Instruction Fields:

`rrr` - destination register field:

```
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]
```

`sss` - source register field:

```
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]
```

# ADDI

## Add with Immediate value

**Operation:**

```
GReg[r] ← GReg[r] + immediate
T ← (GReg[r] == 0)
```

**Assembler**
Syntax: `addi r,immediate`

Example: `addi 6,112`
to ADD GReg[6] and decimal value 112 and store the result in GReg[6]

CPU Flags: T

Cycles: 1

Description: Add a zero-extended immediate value to a General Register; stores the result in the General Register. The flag T is set when the result of the operation is zero; otherwise, it is cleared. The immediate value is the low-order byte of the instruction.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | r | r | r | i | i | i | i | i | i | i | i |

Instruction Fields:

rrr - register field:
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]

iiiiiiii - immediate value:
00000000 - 0
00000001 - 1
...
11111110 - 254
11111111 - 255

# AND Logical AND

**Operation:**

```
GReg[r] ← GReg[s] & GReg[r]
```

**Assembler**
Syntax: `and r,s`

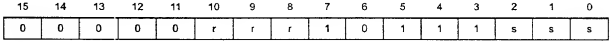
Example: `and 1,2`
to AND GReg[1] and GReg[2] and store the result in GReg[1]

CPU Flags: Unaffected

Cycles: 1

Description: Performs the AND of the source General Register s and the destination General Register r, and stores the result in the destination General Register r.

Instruction Format:



| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | r | r | r | 1 | 0 | 1 | 1 | 1 | s | s | s |

Instruction Fields:

`rrr` - destination register field:
`000` - GReg[0]
`001` - GReg[1]
`010` - GReg[2]
`011` - GReg[3]
`100` - GReg[4]
`101` - GReg[5]
`110` - GReg[6]
`111` - GReg[7]

`sss` - source register field:
`000` - GReg[0]
`001` - GReg[1]
`010` - GReg[2]
`011` - GReg[3]
`100` - GReg[4]
`101` - GReg[5]
`110` - GReg[6]
`111` - GReg[7]

# ANDI

## Logical AND with Immediate value

**Operation:**

```
GReg[r] ← GReg[r] & immediate
```

**Assembler**
Syntax: `andi r,immediate`

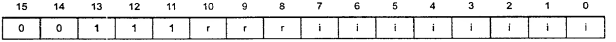
Example: `andi 7,45`
to AND GReg[7] and decimal value 45 and store the result in GReg[7]

CPU Flags: unaffected

Cycles: 1

Description: Performs an AND between a zero-extended immediate value and a General Register; stores the result in the General Register. The immediate value is the low-order byte of the instruction.

Instruction Format:



| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | r | r | r | i | i | i | i | i | i | i | i |

Instruction Fields:

`rrr` - register field:
`000` - GReg[0]
`001` - GReg[1]
`010` - GReg[2]
`011` - GReg[3]
`100` - GReg[4]
`101` - GReg[5]
`110` - GReg[6]
`111` - GReg[7]

`iiiiiiii` - immediate value:
`00000000` - 0
`00000001` - 1
...
`11111110` - 254
`11111111` - 255

# ANDN

## Logical AND NOT

**Operation:**

```
GReg[r] ← ~GReg[s] & GReg[r]
```

**Assembler**
Syntax: `andn r,s`

Example: `andn 3,4`
to AND GReg[3] and NOT GReg[4] (bit inverted) and store the result in GReg[3]

CPU Flags: Unaffected

Cycles: 1

Description: Performs the AND of the negation of the source General Register s and the destination General Register r, and stores the result in the destination General Register r.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | r | r | r | 1 | 0 | 1 | 1 | 0 | s | s | s |

Instruction Fields:

rrr - destination register field:
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]

sss - source register field:
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]

# ANDNI

## Logical AND with Negated Immediate value

**Operation:**

```
GReg[r] ← GReg[r] & ~immediate
```

**Assembler**
Syntax: `andni r,immediate`

Example: `andni 0,2`
to AND GReg[0] and decimal value -3 (inverted 32-bit value 2) and store the result in GReg[0]

CPU Flags: unaffected

Cycles: 1

Description: Performs an AND between the negation of a zero-extended immediate value and a General Register; stores the result in the General Register. The immediate value is the low-order byte of the instruction.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | r | r | r | i | i | i | i | i | i | i | i |

Instruction Fields:

rrr - register field:
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]

iiiiiiii - immediate value:
00000000 - 0
00000001 - 1
...
11111110 - 254
11111111 - 255

# ASR1 Arithmetic Shift Right by 1 Bit

**Operation:**

`GReg[r]:{b31,b30,...,b1,b0}` ← `GReg[r]:{b31,b31,b30,...,b1}`

**Assembler**
Syntax: `asr1 r`

Example: asr1 3
to divide by 2 the signed value of GReg[3] and store the result in GReg[3]

CPU Flags: Unaffected

Cycles: 1

Description: Shift the bits of any General Register to the right and keep the same sign: the left bit (bit 31) is kept untouched.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | r | r | r | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |

Instruction Fields:

rrr - register field:
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]

# BCLRI

## Bit Clear Immediate

**Operation:**

```
GReg[r]:{b31,...,b(i+1),0,b(i-1),...,b0} ←
GReg[r]:{b31,...,b(i+1),b(i),b(i-1),...,b0}
```

**Assembler**
Syntax: `bclri r,i`

Example: `bclri 1,12`
to clear bit 12 in GReg[1]

CPU Flags: Unaffected

Cycles: 1

Description: Clear the bit of register r specified by the immediate field

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | r | r | r | 0 | 0 | 1 | i | i | i | i | i |

Instruction Fields:

rrr - register field:
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]

# BDF

## Conditional Branch if Destination Fault

**Operation:**

```
if (DF == 1) PC ← PC + 1 + displacement else PC ← PC + 1
```

**Assembler**
Syntax: `bdf label`

Example: bdf LLL
to jump to LLL if DF is set, or go to the next instruction if DF is cleared; the displacement value is calculated by the assembler

CPU Flags: Unaffected

Cycles: 2 when the branch is done, 1 otherwise

Description: Conditional branch: if flag DF is set, jump to the new address that is calculated by adding the sign-extended 8-bit displacement to the next PC address. If flag DF is cleared, no jump is performed: the next instruction is located at the next PC address.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | p | p | p | p | p | p | p | p |

Instruction Fields:

pppppppp - signed displacement field:
00000000 - 0
00000001 - 1
...
01111110 - 126
01111111 - 127
10000000 - -128
10000001 - -127
...
11111110 - -2
11111111 - -1

# BF

## Conditional Branch if False

**Operation:**

```
if (T == 0) PC ← PC + 1 + displacement
else PC ← PC + 1
```

**Assembler**
Syntax: `bf label`

Example: `bf LLL`
to jump to LLL if T is cleared, or go to the next instruction if T is set; the displacement value is calculated by the assembler

CPU Flags: Unaffected

Cycles: 2 when the branch is done, 1 otherwise

Description: Conditional branch: if flag T is cleared, jump to the new address that is calculated by adding the sign-extended 8-bit displacement to the next PC address. If flag T is set, no jump is performed: the next instruction is located at the next PC address.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | p | p | p | p | p | p | p | p |

Instruction Fields:

pppppppp - signed displacement field:
00000000 - 0
00000001 - 1
...
01111110 - 126
01111111 - 127
10000000 - -128
10000001 - -127
...
11111110 - -2
11111111 - -1

# BSETI

## Bit Set Immediate

**Operation:**

```
GReg[r]:{b31,...,b(i+1),1,b(i-1),...,b0} ←
GReg[r]:{b31,...,b(i+1),b(i),b(i-1),...,b0}
```

**Assembler**

```
Syntax:    bseti r,i
```

Example: bseti 6,5
to set bit 5 in GReg[6]

CPU Flags: Unaffected

Cycles: 1

Description: Sets bit number i in the selected General Register.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | r | r | r | 0 | 1 | 0 | i | i | i | i | i |

Instruction Fields:

rrr - register field:
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]

iii - bit number field:
00000 - bit 0
00001 - bit 1
...
11110 - bit 30
11111 - bit 31

# BSF

## Conditional Branch if Source Fault

**Operation:**

```
if (SF == 1) PC ← PC + 1 + displacement
else PC ← PC + 1
```

**Assembler**

Syntax: `bsf label`

Example: `bsf LLL`
to jump to LLL if SF is set, or go to the next instruction if SF is cleared; the displacement value is calculated by the assembler

CPU Flags: Unaffected

Cycles: 2 when the branch is done, 1 otherwise

Description: Conditional branch: if flag SF is set, jump to the new address that is calculated by adding the sign-extended 8-bit displacement to the next PC address. If flag SF is cleared, no jump is performed: the next instruction is located at the next PC address.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | p | p | p | p | p | p | p | p |

Instruction Fields:

pppppppp - signed displacement field:
00000000 - 0
00000001 - 1
...
01111110 - 126
01111111 - 127
10000000 - -128
10000001 - -127
...
11111110 - -2
11111111 - -1

# BT Conditional Branch if True

**Operation:**

```
if (T == 1) PC ← PC + 1 + displacement
else PC ← PC + 1
```

**Assembler**
Syntax: `bt label`

Example: bt LLL
to jump to LLL if T is set, or go to the next instruction if T is cleared; the displacement value is calculated by the assembler

CPU Flags: Unaffected

Cycles: 2 when the branch is done, 1 otherwise

Description: Conditional branch: if flag T is set, jump to the new address that is calculated by adding the sign-extended 8-bit displacement to the next PC address. If flag T is cleared, no jump is performed: the next instruction is located at the next PC address.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | p | p | p | p | p | p | p | p |

Instruction Fields:

pppppppp - signed displacement field:
00000000 - 0
00000001 - 1
...
01111110 - 126
01111111 - 127
10000000 - -128
10000001 - -127
...
11111110 - -2
11111111 - -1

# BTSTI Bit Test immediate

**Operation:**

T ← GReg[r]:b(i)

**Assembler**

Syntax: `btsti r,i`

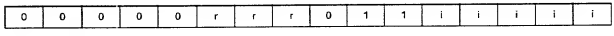
Example: `btsti 2,29`
to test bit 29 in GReg[2] and copy its value in flag T

CPU Flags: T

Cycles: 1

Description: T is loaded with the value of bit number i from the selected General Register.

Instruction Format:




Instruction Fields:

rrr - register field:
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]

iii - bit number field:
00000 - bit 0
00001 - bit 1
...
11110 - bit 30
11111 - bit 31

# CLRF Clear CPU flags

**Operation:**

```
if (ff%2 == 0) SF ← 0
if (ff/2 == 0) DF ← 0
```

**Assembler**
Syntax: `clrf ff`

Example: clrf 2
to clear flag SF and keep flag DF unchanged

CPU Flags: SF, DF

Cycles: 1

Description: Clears a selection of the CPU fault flags: SF, DF, both SF and DF or none can be cleared.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | f | f | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |

Instruction Fields:

`ff` - flags field:
`00` - clear SF and clear DF
`01` - clear DF
`10` - clear SF
`11` - no clear

# CMPEQ

## Compare for Equal

**Operation:**

T ← (GReg[s] == GReg[r])

**Assembler**

Syntax: `cmpeq r,s`

Example: `cmpeq 7,5`
to compare GReg[7] and GReg[5] and set flag T if they are equal

CPU Flags: T

Cycles: 1

Description: Substracts the destination General Register r from the source General Register s, and sets T if the result is zero, clears T if the result is not zero.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | r | r | r | 1 | 1 | 0 | 0 | 1 | s | s | s |

Instruction Fields:

`rrr` - destination register field:
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]

`sss` - source register field:
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]

# CMPEQI Compare with Immediate for Equal

**Operation:**

```
T ← (GReg[r] == immediate)
```

**Assembler**
Syntax: `cmpeqi r,immediate`

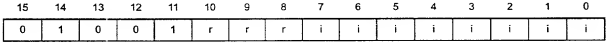
Example: `cmpeqi 2,13`
to compare GReg[2] and decimal value 13 and set flag T if they are equal

CPU Flags: T

Cycles: 1

Description: Subtracts the zero-extended immediate value from the General Register, and sets T if the result is zero, clears T if the result is not zero. The immediate value is the low-order byte of the instruction.

Instruction Format:



| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 1 | r | r | r | i | i | i | i | i | i | i | i |

Instruction Fields:

`rrr` - register field:
`000` - GReg[0]
`001` - GReg[1]
`010` - GReg[2]
`011` - GReg[3]
`100` - GReg[4]
`101` - GReg[5]
`110` - GReg[6]
`111` - GReg[7]

`iiiiiiii` - immediate value:
`00000000` - 0
`00000001` - 1
...
`11111110` - 254
`11111111` - 255

# CMPHS

## Compare for Higher or Same

**Operation:**

T ← (GReg[r] ≥ GReg[s])

**Assembler**
Syntax: cmphs r,s

Example: cmphs 0,1
to compare GReg[0] and GReg[1] and set flag T if GReg[0] is higher than or equal to GReg[1]

CPU Flags: T

Cycles: 1

Description: Compares the destination General Register r and the source General Register s, and sets T if the destination General Register r is higher than or equal to the source General Register s, clears T otherwise. The comparison is unsigned.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | r | r | r | 1 | 1 | 0 | 1 | 1 | s | s | s |

Instruction Fields:

rrr - destination register field:
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]

sss - source register field:
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]

# CMPLT

## Compare for Less Than

**Operation:**

T ← (GReg[r] < GReg[s])

**Assembler**

Syntax: `cmplt r,s`

Example: `cmplt 7,4`
to compare GReg[7] and GReg[4] and set flag T if GReg[7] is lower than GReg[4]

CPU Flags: T

Cycles: 1

Description: Compares the destination General Register r and the source General Register s, and sets T if the destination General Register r is lower than the source General Register s, clears T otherwise. The comparison is signed.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | r | r | r | 1 | 1 | 0 | 1 | 0 | s | s | s |

Instruction Fields:

`rrr` - destination register field:
`000` - GReg[0]
`001` - GReg[1]
`010` - GReg[2]
`011` - GReg[3]
`100` - GReg[4]
`101` - GReg[5]
`110` - GReg[6]
`111` - GReg[7]

`sss` - source register field:
`000` - GReg[0]
`001` - GReg[1]
`010` - GReg[2]
`011` - GReg[3]
`100` - GReg[4]
`101` - GReg[5]
`110` - GReg[6]
`111` - GReg[7]

# DONE DONE, Yield

**Operation:**

```
if (jjj&6 == 2) HE[CCR] ← 0
if (jjj == 3) HE[CCR] ← 1
if (jjj == 4) EP[CCR] ← 0
if (jjj&6 == 6) DE[CCR] ← 0
if (jjj == 7) DE[CCR] ← 1
if ((jjj == 0) && (NCP > CCP)) CCR ← NCR
else if ((jjj == 1) && (NCP >= CCP)) CCR ← NCR
else CCR ← NCR
shPC ← {SF,RPC,T,PC}
shLoop ← {LM,EFC,DF,SPC}
shGReg0 ← GReg[0]
(CCR stands for Current Channel Register; NCR stands for Next Channel Register)
```

**Assembler**

Syntax: `done jjj`

Example: `done 3`
to clear HE bit for the current channel, send an interrupt to the Host for the current channel and reschedule

CPU Flags: Unaffected

Cycles: 47 if a context switch is done, 1 otherwise

Description: Clears one of the channel enabling bits (HE, EP or DE for the corresponding channel number) if required, sends an interrupt to the corresponding CPU by setting the appropriate flag if required (HI or DI for the corresponding channel number), and reschedule according to the mode and the NCP (Next Channel Priority) and CCP (Current Channel Priority) values. According to the scheduling decision, the NCR (Next Channel Register) is copied to the CCR (Current Channel Register) and channel contexts are switched.
If several channels with the same highest priority are pending, they are ordered by their number from 31 down to 0: the higher number will be selected (i.e. channel 26 is selected if channels 3, 12, 14 and 26 with the same highest priority are pending).
If no flag is modified, the reschedule can allow the replacement of the current channel by another channel with a priority strictly greater than the current channel priority (yield); or it can allow the replacement of the current channel by another channel with a priority greater than or equal to the current channel priority (yieldge). In the latter case, the selected channel will always be the first one with the same priority, starting from channel number 31 down to channel 0 (the current channel does not belong to the set of selectable channels).

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | j | j | j | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Instruction Fields:

`jjj` - Channel Flags field:
`000` - no channel flags affected: reschedule only if the next channel priority is greater than current channel priority (yield)
`001` - no channel flags affected: reschedule only if the next channel priority is greater than

or equal to the current channel priority (yieldge)
010 - clear HE for the current channel and reschedule
011 - clear HE, set HI for the current channel and reschedule
100 - clear EP for the current channel and reschedule
101 - reserved for debug to copy relevant registers into their shadows
110 - clear DE for the current channel and reschedule
111 - clear DE, set DI for the current channel and reschedule

# ILLEGAL

## ILLEGAL instruction

**Operation:**

PC ← 0001

**Assembler**
Syntax: illegal

CPU Flags: Unaffected

Cycles: 2

Description: Jumps to the Illegal instruction routine located at address 0001. All unauthorized instructions result in an Illegal instruction behavior; however, the ILLEGAL instruction must be used to guarantee software compatibility with future versions of the IPCM.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |

Instruction Fields:

# JMP

## Unconditional Jump Immediate

**Operation:**

PC ← absolute_address

**Assembler**

Syntax: `jmp label`

Example: `jmp LBL`
the assembler translates the label to the exact address

CPU Flags: Unaffected

Cycles: 2

Description: Jumps to the absolute address contained the lower 14 bits of the instruction (the PC is a 14-bit register).

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | a | a | a | a | a | a | a | a | a | a | a | a | a | a |

Instruction Fields:

aaaaaaaaaaaaaa - address field:
00000000000000 - 0
00000000000001 - 1
...
11111111111110 - 16382
11111111111111 - 16383

# JMPR

## Unconditional Jump

**Operation:**

PC ← GReg[r]

**Assembler**

Syntax: `jmpr r`

Example: `jmpr 0`
to jump to address stored in GReg[0]

CPU Flags: Unaffected

Cycles: 2

Description: Jumps to the absolute address contained in a General Register.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | r | r | r | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |

Instruction Fields:

rrr - register field:
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]

# JSR

## Unconditional Jump to Subroutine Immediate

**Operation:**

```
RPC ← PC + 1
PC ← absolute_address
```

**Assembler**
Syntax: `jsrr r`

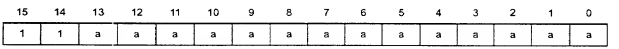
Example: `jsr LLL`
jumps to subroutine starting at LLL; the assembler translates the label to exact address

CPU Flags: Unaffected

Cycles: 2

Description: Jumps to the subroutine located at the absolute address contained the lower 14 bits of the instruction (the PC is a 14-bit register).

Instruction Format:



| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | a | a | a | a | a | a | a | a | a | a | a | a | a | a |

Instruction Fields:

aaaaaaaaaaaaaa - address field:
00000000000000 - 0
00000000000001 - 1
...
11111111111110 - 16382
11111111111111 - 16383

# JSRR

## Unconditional Jump to Subroutine

**Operation:**

```
RPC ← PC + 1
PC ← GReg[r]
```

**Assembler**

Syntax: `jsrr r`

Example: `jsrr 5`
jumps to subroutine located at address stored in GReg[5]

CPU Flags: Unaffected

Cycles: 2

Description: Jumps to the subroutine at address contained in a General Register

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | r | r | r | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |

Instruction Fields:

rrr - register field:

```
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]
```

# LD Load Register

**Operation:**

```
GReg[r] ← [GReg[b] + displacement]
if (transfer_error) SF ← 1
else SF ← 0
```

**Assembler**

Syntax: `ld r,(b,displacement)`

Example: `ld 1,(2,23)`
loads data into GReg[1]; the data is located at address obtained by adding decimal value 23 to GReg[2]

CPU Flags: SF

Cycles: 2+n where n is 0 for ROM, RAM or memory mapped registers, and n is the number of wait-states of the peripheral for a peripheral access

Description: Adds a 5-bit zero-extended displacement to a base address in General Register b; the result is the address of the data to fetch on the DM bus. The data received from the bus is stored in the destination Genera. Register r. If an error occurs during the transfer, the flag SF is set, else it is cleared.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | r | r | r | d | d | d | d | d | b | b | b |

Instruction Fields:

rrr - destination register field:
000 - GReg[0]
001 - GReg[1]
...
111 - GReg[7]

bbb - base address register field:
000 - GReg[0]
001 - GReg[1]
...
110 - GReg[6]
111 - GReg[7]

ddddd - displacement value:
00000 - 0
00001 - 1
...
11110 - 30
11111 - 31

# LDF

## Load Register from Functional Unit

**Operation:**

```
GReg[r] ← [fu_address]
if (transfer_error) SF ← 1
else SF ← 0
fu_address is an 8-bit field:
                7:3    fureg
                2      fetch
                1:0    size
```

**Assembler**

Syntax: `ldf r,fu_address`

Example: `ldf 0,18`
loads data coming from the Host DMA register MD into GReg[0]; it is a 16-bit access with no prefetch

CPU Flags: SF

Cycles: 1+n where n is the number of wait-states that may be inserted by the functional unit

Description: Sends an 8-bit address on the Functional Unit Bus (FU bus) and stores the data received from the bus in the destination General Register r. If an error occurs during the transfer, the flag SF is set, else it is cleared.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 | r | r | r | u | u | u | u | u | u | u | u |

Instruction Fields:

rrr - destination register field:
000 - GReg[0]
001 - GReg[1]
...
110 - GReg[6]
111 - GReg[7]

uuuuuuuu - functional unit address field (x is a don't-care bit):
00000xxx- read MA (no side effect)
00010x00- read MD (no side effect)
00010p01- read MD: 8-bit access
MA ← MA + 1
prefetch if ((p == 1) && (MA%4 == 0))
00010p10- read MD: 16-bit access
MA ← MA + 2
prefetch if ((p == 1) && (MA%4 == 0))
00010p11- read MD: 32-bit access
MA ← MA + 4

```
prefetch if ((p == 1) && (MA%4 == 0))
00011xxx- read MS (no side effect)
00100xxx- read DA (no side effect)
00110x00- read DD (no side effect)
00110p01- read DD: 8-bit access
DA ← DA+1
prefetch if ((p == 1) && (DA%4 == 0))
00110p10- read DD: 16-bit access
DA ← DA+2
prefetch if ((p == 1) && (DA%4 == 0))
00110p11- read DD: 32-bit access
DA ← DA+4
prefetch if ((p == 1) && (DA%4 == 0))
00111xxx- read DS (no side effect)
01000xxx- read CA (no side effect)
01001xxx- read CS (right aligned, no side effect)
```

# LDI

## Load Register with Immediate value

**Operation:**

GReg[r] ← immediate

**Assembler**

Syntax: `ldi r,immediate`

Example: `ldi 6,1`
loads decimal value 1 into GReg[6]

CPU Flags: Unaffected

Cycles: 1

Description: Stores a zero-extended immediate value in a General Register. The immediate value is the low-order byte of the instruction.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | r | r | r | i | i | i | i | i | i | i | i |

Instruction Fields:

rrr - register field:
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]

iiiiiiii - immediate value:
00000000 - 0
00000001 - 1
...
11111110 - 254
11111111 - 255

# LDRPC

## Load from RPC to Register

**Operation:**

GReg[r] ← RPC

**Assembler**

Syntax: `ldrpc r`

Example: `ldrpc 3`
copies RPC to GReg[3]

CPU Flags: Unaffected

Cycles: 1

Description: Stores the contents of the RPC in a General Register. That instruction may be used to have more than one level of subroutines.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | r | r | r | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |

Instruction Fields:

rrr - register field:

000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]

# LOOP Hardware Loop

**Operation:**

```
if (ff%2 == 0) SF ← 0
if (ff/2 == 0) DF ← 0
if ((GReg[0] == 0) || (SF == 1) || (DF == 1))
                    PC ← PC + loop_size + 1
else {
                    SPC ← PC + 1
                    EPC ← PC + loop_size + 1
                    LM ← 1
                    PC ← PC + 1
}
during each instruction execution in the loop:
if ((SF == 1) || (DF == 1)) {
                    LM ← 0
                    PC ← EPC
}
else if ((PC + 1) == EPC) {
                    GReg[0] ← GReg[0] - 1
                    if (GReg[0] == 0) {
                          LM ← 0
                          PC ← EPC
                    }
                    else PC ← SPC
}
else PC ← nextPC(instruction)
after the execution of the last instruction of the loop body:
if (GReg[0] == 0)
                    T ← 1
else
                    T ← 0
```

**Assembler**

Syntax: `loop n{,ff}`

Example: `loop 3,0`
execu.es GReg[0] times the instructions comprised between PC+1 and PC+3 (included); both SF and DF flags are cleared before starting the loop; when omitted, the `ff` field will be set to 0 (clearing both SF and DF)

CPU Flags: LM, T

Cycles: 2 when the loop count (GReg[0]) is 0 or SF or DF is set at loop start, 1+1 when the loop starts but exits abnormally (SF or DF set inside the loop which adds 1 cycle to the offending load or store to jump to EPC), 1 when the loop is executed normally

Description: The LOOP instruction executes several times a sequence of instructions. The number of times is given by the contents of GReg[0] that is the loop counter. That means the IPCM will jump to the first instruction after the end of the loop if GReg[0] value is 0; if not, the IPCM enters loop mode: it sets the LM flag that will only be reset once the last instruction of the last loop is executed. The instructions in the loop will be executed GReg[0] times.
The management of fault flags (SF and DF) is as follows: when entering the hardware loop, SF and DF can be cleared according to the ff field of the instruction; after that operation, if any flag is still set, the loop will not be executed: the IPCM will jump to the first instruction after the end of the loop without entering loop mode. During the execution of the loop, if any fault flag is set by a LD, LDF, ST or STF instruction, the IPCM will immediately exit loop mode and jump to the first instruction after the end of the loop. In that case, GReg0 is not decremented for that last piece of the loop body execution (this is even the case if the SF or DF flag is set at the last instruction of the loop body).
The T flag reflects the state of GReg[0] after the end of the loop, which is an indicator of the complete

execution of the loop: if the loop exited because of an error (SF or DF set), GReg[0] will not be 0 at the end of the loop, hence T will be cleared; if the loop execution went well, GReg[0] will be 0 at the end of the loop, hence T will be set. The boundary case when a source or destination fault occurs at the last instruction of the last loop is considered as an anticipated exit of the loop, which causes the T flag to be cleared. If the last instruction executed before leaving the hardware loop also tries to modify the T flag, the flag is updated according to the value of GReg[0], NOT according to the result of the last executed instruction.

Limitations: Jump instructions (JMP, JMPR, JSR, JSRR, BF, BT, BSF, BDF) are not allowed inside the hardware loop (we are working on this: some jumps will be allowed in the future but beware of boundary cases - the exact behavior of the hardware will be completely specified in all the cases).
- GReg[0] cannot be written to inside the hardware loop (it can be read).
- the empty loop (0 instruction in the body) is forbidden.
- if GReg[0] == 0 at the start of the loop, which causes a jump to EPC, the T flag is not updated (we are also working on this: the intention is to have the T flag set).

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | 0 | f | f | n | n | n | n | n | n | n | n |

Instruction Fields:

ff - flags field:
00 - clear SF and clear DF
01 - clear DF
10 - clear SF
11 - no clear

nnnnnnnn - loop size
00000000 - empty loop: forbidden value
00000001 - 1 instruction in the loop
00000010 - 2 instructions in the loop
...
11111111 - 255 instructions in the loop

# LSL1

## Logical Shift Left by 1 Bit

**Operation:**

`GReg[r]:{b30,...,b1,b0,0} ← GReg[r]:{b31,b30,...,b1,b0}`

**Assembler**
Syntax: `lsl1 r`

Example: `lsl1 2`
multiplies by 2 the value in GReg[2]

CPU Flags: Unaffected

Cycles: 1

Description: Shift the bits of any General Register to the left. The right bit (bit 0) is set to 0. No overflow is detected by the hardware.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | r | r | r | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |

Instruction Fields:

rrr - register field:
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]

# LSR1

## Logical Shift Right by 1 Bit

**Operation:**

GReg[r]:{0,b31,b30,...,b1} ← GReg[r]:{b31,b30,...,b1,b0}

Syntax: `lsr1 r`

Example: `lsr1 4`
divides by 2 the unsigned value contained in GReg[4]

CPU Flags: Unaffected

Cycles: 1

Description: Shift the bits of any General Register to the right. The left bit (bit 31) is set to 0.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | r | r | r | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |

Instruction Fields:

rrr - register field:
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]

# MOV

## Logical Move

**Operation:**

`GReg[r] ← GReg[s]`

**Assembler**
Syntax: `mov r,s`

Example: `mov 4,0`
copies GReg[0] to GReg[4]

CPU Flags: Unaffected

Cycles: 1

Description: Move the contents of the source General Register s to the destination General Register r.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | r | r | r | 1 | 0 | 0 | 0 | 1 | s | s | s |

Instruction Fields:

rrr - destination register field:
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]

sss - source register field:
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]

# NOTIFY

## Notify to MCU/DSP

**Operation:**

```
if (jjj&4 == 0) {
                if (jjj&2 == 2) HE[CCR] ← 0
                if (jjj&1 == 1) HI[CCR] ← 1
}
else if (jjj == 4) EP[CCR] ← 0
else {
                if (jjj&2 == 2) DE[CCR] ← 0
                if (jjj&1 == 1) DI[CCR] ← 1
}
(CCR stands for Current Channel Register)
```

**Assembler**

Syntax: `notify jjj`

Example: `notify 7`
clears the DE bit for the current channel and sends an interrupt to the DSP for the current channel

CPU Flags: Unaffected

Cycles: 1

Description: Clears one of the channel enabling bits (HE, EP or DE for the corresponding channel number) if required, sends an interrupt to the corresponding CPU by setting the appropriate flag if required (HI or DI for the corresponding channel number).

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | j | j | j | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

Instruction Fields:

`jjj` - Channel Flags field:
`000` - unused
`001` - set HI for the current channel
`010` - clear HE for the current channel
`011` - clear HE, set HI for the current channel
`100` - clear EP for the current channel
`101` - set DI for the current channel
`110` - clear DE for the current channel
`111` - clear DE, set DI for the current channel

# OR Logical OR

**Operation:**

```
GReg[r] ← GReg[s] | GReg[r]
```

**Assembler Syntax:** `or r,s`

Example: `or 3,6`
ORs GReg[3] and GReg[6] and stores the result in GReg[6]

CPU Flags: Unaffected

Cycles: 1

Description: Performs the OR of the source General Register s and the destination General Register r, and stores the result in the destination General Register r.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | r | r | r | 1 | 0 | 1 | 0 | 1 | s | s | s |

Instruction Fields:

`rrr` - destination register field:
`000` - GReg[0]
`001` - GReg[1]
`010` - GReg[2]
`011` - GReg[3]
`100` - GReg[4]
`101` - GReg[5]
`110` - GReg[6]
`111` - GReg[7]

`sss` - source register field:
`000` - GReg[0]
`001` - GReg[1]
`010` - GReg[2]
`011` - GReg[3]
`100` - GReg[4]
`101` - GReg[5]
`110` - GReg[6]
`111` - GReg[7]

# ORI Logical OR with Immediate value

**Operation:**

```
GReg[r] ← GReg[r] | immediate
```

**Assembler**
Syntax: `ori r,immediate`

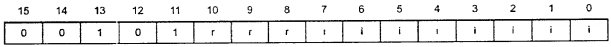
Example: `ori 1,56`
ORs GReg[1] and the decimal value 56 and stores the result in GReg[1]

CPU Flags: unaffected

Cycles: 1

Description: Performs an OR between a zero-extended immediate value and a General Register; stores the result in the General Register. The immediate value is the low-order byte of the instruction.

Instruction Format:




Instruction Fields:

`rrr` - register field:
`000` - GReg[0]
`001` - GReg[1]
`010` - GReg[2]
`011` - GReg[3]
`100` - GReg[4]
`101` - GReg[5]
`110` - GReg[6]
`111` - GReg[7]

`iiiiiiii` - immediate value:
`00000000` - 0
`00000001` - 1
...
`11111110` - 254
`11111111` - 255

# RET Return from subroutine

**Operation:**

PC ← RPC

**Assembler**
Syntax: `ret`

CPU Flags: Unaffected

Cycles: 2

Description: Return from subroutine.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

Instruction Fields:

# REVB

## Reverse Byte order

**Operation:**

`GReg[r]:{B3,B2,B1,B0} ← GReg[r]:{B0,B1,B2,B3}`

**Assembler**
Syntax: `revb r`

Example: `revb 5`
reverses bytes order in GReg[5]

CPU Flags: Unaffected

Cycles: 1

Description: Reverse the byte order of any General Register.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | r | r | r | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

Instruction Fields:

rrr - register field:
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]

# REVBLO Reverse Low Order Bytes

**Operation:**

`GReg[r]:{B3,B2,B0,B1}` ← `GReg[r]:{B3,B2,B1,B0}`

**Assembler**
Syntax: `revblo r`

Example: `revblo 0`
reverses low order bytes in GReg[0]

CPU Flags: Unaffected

Cycles: 1

Description: Reverse both low order bytes of any General Register.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | r | r | r | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |

Instruction Fields:

rrr - register field:
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]

# ROR1

## Rotate Right by 1 bit

**Operation:**

`GReg[r]:{b0,b31,b30,...,b1} ← GReg[r]:{b31,b30,...,b1,b0}`

**Assembler**

Syntax: `ror1 r`

Example: `ror1 3`
rotates bits to the right in GReg[3]

CPU Flags: Unaffected

Cycles: 1

Description: Rotate the bits of any General Register to the right.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | r | r | r | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |

Instruction Fields:

rrr - register field:
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]

# RORB

## Rotate Right by 1 Byte

**Operation:**

`GReg[r]:{B0,B3,32,B1}` ← `GReg[r]:{B3,B2,B1,B0}`

**Assembler**
Syntax: `rorb r`

Example: `rorb 2`
rotates bytes to the right in GReg[2]

CPU Flags: Unaffected

Cycles: 1

Description: Rotate the bytes of any General Register to the right.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | r | r | r | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |

Instruction Fields:

`rrr` - register field:
`000` - `GReg[0]`
`001` - `GReg[1]`
`010` - `GReg[2]`
`011` - `GReg[3]`
`100` - `GReg[4]`
`101` - `GReg[5]`
`110` - `GReg[6]`
`111` - `GReg[7]`

# SOFTBKPT Software Breakpoint

**Operation:**

Stops the current script and enters debug mode

**Assembler**
Syntax: softbkpt

CPU Flags: Unaffected

Cycles:

Description:

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |

Instruction Fields:

# ST Store Register

**Operation:**

```
[GReg[b] + displacement] ← GReg[r]
if (transfer_error) DF ← 1
else DF ← 0
```

**Assembler**
Syntax: `st r,(b,displacement)`

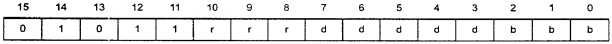
Example: `st 7,(0,9)`
stores the value from GReg[7] into memory at address obtained by adding decimal value 9 to GReg[0]

CPU Flags: DF

Cycles: 2+n where n is 0 for ROM, RAM or memory mapped registers, and n is the number of wait-states of the peripheral for a peripheral access

Description: Adds a 5-bit zero-extended displacement to a base address in General Register b; the result is the address of the data to store on the DM bus. The data sent on the bus comes from the source General Register r. If an error occurs during the transfer, the flag DF is set, else it is cleared.

Instruction Format:



| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 1 | r | r | r | d | d | d | d | d | b | b | b |

Instruction Fields:

rrr - source register field:
000 - GReg[0]
001 - GReg[1]
...
110 - GReg[6]
111 - GReg[7]

bbb - base address register field:
000 - GReg[0]
001 - GReg[1]
...
110 - GReg[6]
111 - GReg[7]

ddddd - displacement value:
00000 - 0
00001 - 1
...
11110 - 30
11111 - 31

# STF

## Store Register in Functional Unit

**Operation:**

```
[fu_address] ← GReg[r]
if (transfer_error) DF ← 1
else DF ← 0
fu_address is an 8-bit field:
                7:3    fureg
                2      fetch / flush
                1:0    size
```

**Assembler**
Syntax: `stf r,fu_address`

Example: `stf 3,55`
stores the 32-bit contents of GReg[3] to the DSP DMA register DD; waits until the flush to external DSP memory is completed

CPU Flags: DF

Cycles: 1+n where n is the number of wait-states that may be inserted by the functional unit

Description: Sends an 8-bit address on the Functional Unit Bus (FU bus) and sends the contents of the source General Register r on the bus. If an error occurs during the transfer, the flag DF is set, else it is cleared.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 1 | r | r | r | u | u | u | u | u | u | u | u |

Instruction Fields:

rrr - source register field:
000 - GReg[0]
001 - GReg[1]
...
110 - GReg[6]
111 - GReg[7]

uuuuuuuu - functional unit address field (x is a don't-care bit):
00000x00- write MA (no side effect)
00000p01- write MA
10flush if (MD not empty)
11prefetch if (p == 1)
transfer_error if ((p ==1) && (MD not empty))
00010000- write MD (no side effect)
00010100- no write: flush if MD is not empty
00010f01- write MD: 8-bit access
MA ← MA + 1
flush if ((f == 1) || (MA%4 == 0))

00010f10- write MD: 16-bit access
MA ← MA + 2
flush if ((f == 1) || (MA%4 == 0))
00010f11- write MD: 32-bit access
MA ← MA + 4
flush if ((f == 1) || (MA%4 == 0))
00011xxx- write MS (no side effect)
00100x00- write DA (no side effect)
00100p01- write DA
10flush if (DD not empty)
11prefetch if (p == 1)
00110000- write DD (no side effect)
00110100- no write: flush if DD is not empty
00110f01- write DD: 8-bit access
DA ← DA+1
flush if ((f == 1) || (DA%4 == 0))
00110f10- write DD: 16-bit access
DA ← DA+2
flush if ((f == 1) && (DA%4 == 0))
00110f11- write DD: 32-bit access
DA ← DA+4
flush if ((f == 1) && (DA%4 == 0))
00111xxx- write DS (no side effect)
01000xxx- write CA (no side effect)
01001xx0- write CS (right aligned, no side effect)
01001xx1- write CS: compute CRC with new incoming byte

# SUB Substract

**Operation:**

```
GReg[r] ← GReg[r] - GReg[s]
T ← (GReg[r] == 0)
```

**Assembler**
Syntax: `sub r,s`

Example: `sub 4,7`
SUBstracts GReg[7] from GReg[4] and stores the result in GReg[4]

CPU Flags: T

Cycles: 1

Description: Substracts the source General Register s from the destination General Register r, and stores the result in the destination General Register r. The T flag is set if the result of the operation is 0; it is cleared if the result is not zero.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | r | r | r | 1 | 0 | 1 | 0 | 0 | s | s | s |

Instruction Fields:

rrr - destination register field:
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]

sss - source register field:
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]

# SUBI

## Substract with Immediate

**Operation:**

`GReg[r] ← GReg[r] - immediate T ← (GReg[r] == 0)`

**Assembler**

Syntax: `sub r,immediate`

Example: `sub 1,255`
SUBstracts decimal value 255 from GReg[1] and stores the result in GReg[1]

CPU Flags: T

Cycles: 1

Description: Substracts a zero-extended immediate value from a General Register; stores the result in the General Register. The flag T is set when the result of the operation is zero; otherwise, it is cleared. The immediate value is the low-order byte of the instruction.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | r | r | r | i | i | i | i | i | i | i | i |

Instruction Fields:

rrr - register field:
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]

iiiiiiii - immediate value:
00000000 - 0
00000001 - 1
...
11111110 - 254
11111111 - 255

# TST Test with Zero

**Operation:**

```
T ← ((GReg[s] & GReg[r]) != 0)
```

**Assembler**
Syntax: `tst r,s`

Example: `tst 2,3`
ANDs GReg[2] and GReg[3] and sets T if the result is non-null

CPU Flags: T

Cycles: 1

Description: Performs the AND of the source General Register s and the destination General Register r, and sets T if the result is not zero, clears T if the result is zero.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | r | r | r | 1 | 1 | 0 | 0 | 0 | s | s | s |

Instruction Fields:

`rrr` - destination register field:
`000` - GReg[0]
`001` - GReg[1]
`010` - GReg[2]
`011` - GReg[3]
`100` - GReg[4]
`101` - GReg[5]
`110` - GReg[6]
`111` - GReg[7]

`sss` - source register field:
`000` - GReg[0]
`001` - GReg[1]
`010` - GReg[2]
`011` - GReg[3]
`100` - GReg[4]
`101` - GReg[5]
`110` - GReg[6]
`111` - GReg[7]

# TSTI

## Mask with Zero Immediate

**Operation:**

```
T ← ((GReg[r] & immediate) != 0)
```

**Assembler**
Syntax: `tsti r,immediate`

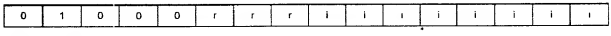
Example: `tsti 5,13`
ANDs GReg[5] and decimal value 13 and sets T if the result is non-null

CPU Flags: T

Cycles: 1

Description: Performs the AND of a zero-extended immediate value and the destination General Register r, and sets T if the result is not zero, clears T if the result is zero. The immediate value is the low-order byte of the instruction.

Instruction Format:



| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | r | r | r | i | i | i | i | i | i | i | i |

Instruction Fields:

rrr - register field:
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]

iiiiiiii - immediate value:
00000000 - 0
00000001 - 1
...
11111110 - 254
11111111 - 255

# XOR Logical Exclusive OR

**Operation:**

`GReg[r] ← GReg[s] ^ GReg[r]`

**Assembler**
Syntax: `xor r,s`

Example: `xor 0,3`
XORs GReg[0] and GReg[3] and stores the result in GReg[0]

CPU Flags: Unaffected

Cycles: 1

Description: Performs the eXclusive OR of the source General Register s and the destination General Register r, and stores the result in the destination General Register r.

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | r | r | r | 1 | 0 | 0 | 1 | 0 | s | s | s |

Instruction Fields:

`rrr` - destination register field:
`000` - GReg[0]
`001` - GReg[1]
`010` - GReg[2]
`011` - GReg[3]
`100` - GReg[4]
`101` - GReg[5]
`110` - GReg[6]
`111` - GReg[7]

`sss` - source register field:
`000` - GReg[0]
`001` - GReg[1]
`010` - GReg[2]
`011` - GReg[3]
`100` - GReg[4]
`101` - GReg[5]
`110` - GReg[6]
`111` - GReg[7]

# XORI

## Exclusive OR with Immediate

**Operation:**

```
GReg[r] ← GReg[r] ^ immediate
```

**Assembler**
Syntax: `xor r,immediate`

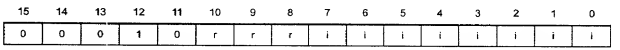
Example: `xor 7,5`
XORs GReg[5] and decimal value 5 and stores the result in GReg[7]

CPU Flags: Unaffected

Cycles: 1

Description: Performs an eXclusive OR between a zero-extended immediate value and a General Register; stores the result in the General Register. The immediate value is the low-order byte of the instruction.

Instruction Format:



| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | r | r | r | i | i | i | i | i | i | i | i |

Instruction Fields:

rrr - register field:
000 - GReg[0]
001 - GReg[1]
010 - GReg[2]
011 - GReg[3]
100 - GReg[4]
101 - GReg[5]
110 - GReg[6]
111 - GReg[7]

iiiiiiii - immediate value:
00000000 - 0
00000001 - 1
...
11111110 - 254
11111111 - 255

# MvShPC2Gr1 Move Data from Shadow PC register to Register 1

**Operation:**

GReg[1] ← ShPCReg

**Assembler**
Syntax: none as this instruction should only be used through the OnCE

CPU Flags: Unaffected

Cycles: 1

Description: Once debug specific instruction. Move the contents of the shadow PC register to the General register[1].

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

Instruction Fields:

# MvShLoop2Gr1 Move Data from Shadow Loop register to Register 1

**Operation:**

GReg[1] ← ShLoopReg

**Assembler**

Syntax: none as this instruction should only be used through the OnCE

CPU Flags: Unaffected

Cycles: 1

Description: Once debug specific instruction. Move the contents of the shadow Loop register to the General register[1].

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

Instruction Fields:

# MvShGr02Gr1 Move Data from Shadow GReg[0] register to Register 1

**Operation:**

GReg[1] ← ShGReg[0]

**Assembler Syntax:** none as this instruction should only be used through the OnCE

CPU Flags: Unaffected

Cycles: 1

Description: Once debug specific instruction. Move the contents of the shadow GReg[0] register to the General register[1 The ShGReg[0] register is used during context switch.

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

Instruction Fields:

# MvGr12ShPC Move Data from Register 1 to Shadow PC register

**Operation:**

ShPCReg ← GReg[1]

**Assembler**
Syntax: none as this instruction should only be used through the OnCE

CPU Flags: Unaffected

Cycles: 1

Description: Once debug specific instruction. Move the contents of the General register[1] to the Shadow PC register.

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

Instruction Fields:

# MvGr12ShLoop Move Data from Register 1 to Shadow Loop register

**Operation:**

ShPCReg ← GReg[1]

**Assembler**
Syntax: none as this instruction should only be used through the OnCE

CPU Flags: Unaffected

Cycles: 1

Description: Once debug specific instruction. Move the contents of the General register[1] to the Shadow Loop register.

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

Instruction Fields:

# reschedule

**Assembler**
Syntax: `reschedule`

CPU Flags: Unaffected

Cycles: 1

Description: Depending on HPPR and HPCR (TestPending), the instruction will either put the ipem core in IDLE mode or continue the context switch subroutine (AndSwitch).

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

Instruction Fields:

# CtxPtrInit --Context Switch specific instruction--

-- THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE--

**Operation:**

GReg[0] ← DM[0x7002]

**Assembler**

Syntax: none as this instruction can not be used outside of the ROM context switch routine.

CPU Flags: Unaffected

Cycles: 1

Description: The base address where all control/general registers will be spilled on execution of a context switch instruction (done or yield) is stored in GReg[0].

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |

Instruction Fields:

# CatchCPtr--Context Switch specific instruction--

-- THIS INSTRUCT ON CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE--

**Operation:**

]

**Assembler**

Syntax: `none as this instruction can not be used outside of the ROM context switch routine.`

CPU Flags: Unaffected

Cycles: 1

Description: The base address where all control/general registers will be spilled on execution of a context switch instruction (done or yield) is stored in GReg[0].

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |

Instruction Fields:

# IdMAstG1 --Context Switch specific instruction--

-- THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE--

**Operation:**

```
GReg[1]->DM[ G[0] + 1 ]
G[0] + 1 -> G[0]
MA -> GReg[1]
```

**Assembler**

Syntax: `none as this instruction can not be used outside of the ROM context switch routine.`

CPU Flags: T

Cycles: 1

Description: store GReg[1] to memory, update GReg[1] with MA and increment address pointer GReg[0].

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

Instruction Fields:

# IdMDstG2--Context Switch specific instruction--

-- THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE--

**Operation:**

```
GReg[2]->DM[ G[0] + 1 ]
G[0] + 1 -> G[0]
MD -> GReg[2]
```

**Assembler**

Syntax: `none as this instruction can not be used outside of the ROM context switch routine.`

CPU Flags: T

Cycles: 1

Description: store C Reg[2] to memory, update GReg[2] with MD and increment address pointer GReg[0].

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

Instruction Fields:

# IdMSstG3--Context Switch specific instruction--

-- THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE--

**Operation:**

```
GReg[3]->DM[ G[0] + 1 ]
G[0] + 1 -> G[0]
MS -> GReg[3]
```

**Assembler**

Syntax: none as this instruction can not be used outside of the ROM context switch routine.

CPU Flags: T

Cycles: 1

Description: store GReg[3] to memory, update GReg[3] with MS and increment address pointer GReg[0].

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

Instruction Fields:

# ldDAstG4--Context Switch specific instruction--

--THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE--

**Operation:**

```
GReg[4]->DM[ G[0] + 1 ]
G[0] + 1 -> G[0]
DA-> GReg[4]
```

**Assembler**

Syntax: `none as this instruction can not be used outside of the ROM context switch routine.`

CPU Flags: T

Cycles: 1

Description: store GReg[4] to memory, update GReg[4] with DA and increment address pointer GReg[0].

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

Instruction Fields:

# ldDDstG5--Context Switch specific instruction--

– THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE–

**Operation:**

```
GReg[5]->DM[ G[0] + 1 ]
G[0] + 1 -> G[0]
DD-> GReg[5]
```

**Assembler**

Syntax: `none as this instruction can not be used outside of the ROM context switch routine.`

CPU Flags: T

Cycles: 1

Description: store GReg[5] to memory, update GReg[5] with DD and increment address pointer GReg[0].

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

Instruction Fields:

# ldDSstG6--Context Switch specific instruction--

-- THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE--

**Operation:**

```
GReg[6]->DM[ G[0] + 1 ]
G[0] + 1 -> G[0]
DS-> GReg[6]
```

**Assembler**

Syntax: `none as this instruction can not be used outside of the ROM context switch routine.`

CPU Flags: T

Cycles: 1

Description: store GReg[6] to memory, update GReg[6] with DD and increment address pointer GReg[0].

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

Instruction Fields:

# LdCAstG7 --Context Switch specific instruction--

-- THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE--

**Operation:**

```
GReg[7]->DM[ G[0] + 1 ]
G[0] + 1 -> G[0]
CA-> GReg[7]
```

**Assembler**

Syntax: `none as this instruction can not be used outside of the ROM context switch routine.`

CPU Flags: T

Cycles: 1

Description: store GReg[7] to memory, update GReg[7] with CA and increment address pointer GReg[0].

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

Instruction Fields:

# stG7mvShPC --Context Switch specific instruction--

-- THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE--

**Operation:**

```
GReg[7]->DM[ G[0] + 1]
G[0] + 1 -> G[0]
ShPCReg -> GReg[7]
```

**Assembler**

Syntax: `none as this instruction can not be used outside of the ROM context switch routine.`

CPU Flags: T

Cycles: 1

Description: store GReg[7] to memory, update GReg[7] with ShPCReg and increment address pointer GReg[0].

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |

Instruction Fields:

# stG7mvShLoop--Context Switch specific instruction--

-- THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE--

**Operation:**

```
GReg[7]->DM[ G[0] + 1]
G[0] + 1 -> G[0]
ShLoopReg -> GReg[7]
```

**Assembler**

Syntax: `none as this instruction can not be used outside of the ROM context switch routine.`

CPU Flags: T

Cycles: 1

Description: store GReg[7] to memory, update GReg[7] with ShLoopReg and increment address pointer GReg[0].

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |

Instruction Fields:

# stG7ldCS--Context Switch specific instruction--

– THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE–

**Operation:**

```
GReg[7]->DM[ G[0] + 1]
G[0] + 1 -> G[0]
CS -> GReg[7]
```

**Assembler**
Syntax: `none as this instruction can not be used outside of the ROM context switch routine.`

CPU Flags: T

Cycles: 1

Description: store GReg[7] to memory, update GReg[7] with CS and increment address pointer GReg[0].

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

Instruction Fields:

# stCAmovShReg02Gr1--Context Switch specific instruction--

-- THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE--

**Operation:**

```
GReg[7]->DM[ G[0] + 1]
ShReg0 -> GReg[1]
```

**Assembler**

Syntax: `none as this instruction can not be used outside of the ROM context switch routine.`

CPU Flags: none

Cycles: 1

Description: store GReg[7] to memory, update GReg[1] with ShReg0.

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |

Instruction Fields:

# TstPendingAndSwitch--Context Switch specific instruction--

-- THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE--

Syntax: none as this instruction can not be used outside of the ROM context switch routine.

CPU Flags: none!

Description: Depending on HPPR and HPCR (TestPending), the instruction will either put the ipem core in IDLE mode or continue the context switch subroutine (AndSwitch).
During same cycle, content of GeneralReg[1] will be stored in Data Memory at address pointed by GeneralReg[0].

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |

Instruction Fields:

# IdFU0inLd0--Context Switch specific instruction--

*– THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE–*

**Operation:**

```
read access DM[G[0]]
G[0] + 1 -> G[0]
```

**Assembler**

Syntax: `none as this instruction can not be used outside of the ROM context switch routine.`

CPU Flags: T

Cycles: 1

Description: Start a read access at GReg[0] address and increments address pointer GReg[0]

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |

Instruction Fields:

# mvFU02G1 --Context Switch specific instruction--

– THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE–

**Operation:**

*(Soft pipeline)* result of former cycle initiated read -> GReg[1]

*(Soft pipeline)* initiate DM[ G[0] ] read access. Data available next cycle.

GReg[0] - 1 -> GReg[0]

**Assembler**

Syntax: none as this instruction can not be used outside of the ROM context switch routine.

CPU Flags: T

Cycles: 1

Description:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |

Instruction Fields:

# ldmfub7 --Context Switch specific instruction--

-- THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE--

**Operation:**

```
GReg[1] -> MA
(Soft pipeline) result of former cycle initiated read -> GReg[1]
(Soft pipeline) initiate DM[ G[0] ] read access. Data available next cycle.
GReg[0] - 1 -> GReg[0]
```

**Assembler**

Syntax: none as this instruction can not be used outside of the ROM context switch routine.

CPU Flags: T

Cycles: 1

Description: .

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |

Instruction Fields:

# ldmfub6--Context Switch specific instruction--

-- THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE--

**Operation:**

```
GReg[1] -> MD
(Soft pipeline) result of former cycle initiated read -> GReg[1]
(Soft pipeline) initiate DM[ G[0] ] read access. Data available next cycle.
GReg[0] - 1 -> GReg[0]
```

**Assembler**

Syntax: none as this instruction can not be used outside of the ROM context switch routine.

CPU Flags: T

Cycles: 1

Description: .

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |

Instruction Fields:

# ldmfub5--Context Switch specific instruction--

-- THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE--

**Operation:**

```
GReg[1] -> MS
(Soft pipeline, result of former cycle initiated read -> GReg[1]
(Soft pipeline, initiate DM[ G[0] ] read access. Data available next cycle.
GReg[0] - 1 -> GReg[0]
```

**Assembler**

Syntax: none as this instruction can not be used outside of the ROM context switch routine.

CPU Flags: T

Cycles: 1

Description: .

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |

Instruction Fields:

# ldmfub4--Context Switch specific instruction--

-- THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE--

**Operation:**

```
GReg[1] -> DA
(Soft pipeline) result of former cycle initiated read -> GReg[1]
(Soft pipeline) initiate DM[ G[0] ] read access. Data available next cycle.
GReg[0] - 1 -> GReg[0]
```

**Assembler**

Syntax: none as this instruction can not be used outside of the ROM context switch routine.

CPU Flags: T

Cycles: 1

Description: .

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |

Instruction Fields:

# ldmfub3 --Context Switch specific instruction--

– THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE–

**Operation:**

```
GReg[1] -> DD
(Soft pipeline) result of former cycle initiated read -> GReg[1]
(Soft pipeline) initiate DM[ G[0] ] read access. Data available next cycle.
GReg[0] - 1 -> GReg[0]
```

**Assembler**

Syntax: none as this instruction can not be used outside of the ROM context switch routine.

CPU Flags: T

Cycles: 1

Description: .

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |

Instruction Fields:

# ldmfub2 --Context Switch specific instruction--

-- THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE--

**Operation:**

```
GReg[1] -> DS
(Soft pipeline) result of former cycle initiated read -> GReg[1]
(Soft pipeline) initiate DM[ G[0] ] read access. Data available next cycle.
GReg[0] - 1 -> GReg[0]
```

**Assembler**

Syntax: none as this instruction can not be used outside of the ROM context switch routine.

CPU Flags: T

Cycles: 1

Description: .

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |

Instruction Fields:

# ldmfub1 --Context Switch specific instruction--

-- THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE--

**Operation:**

```
GReg[1] -> CA
(Soft pipeline) result of former cycle initiated read -> GReg[1]
(Soft pipeline) initiate DM[ G[0] ] read access. Data available next cycle.
GReg[0] - 1 -> GReg[0]
```

**Assembler**

Syntax: none as this instruction can not be used outside of the ROM context switch routine.

CPU Flags: T

Cycles: 1

Description: .

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |

Instruction Fields:

# ldmfub0 --Context Switch specific instruction--

-- THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE--

**Operation:**

```
GReg[1] -> CS
(Soft pipeline) result of former cycle initiated read -> GReg[1]
(Soft pipeline) initiate DM[ G[0] ] read access. Data available next cycle.
GReg[0] - 1 -> GReg[0]
```

**Assembler**

Syntax: none as this instruction can not be used outside of the ROM context switch routine.

CPU Flags: T

Cycles: 1

Description: .

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |

Instruction Fields:

# ldShLoop --Context Switch specific instruction--

-- THIS INSTRUCT ON CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE--

**Operation:**

*(Soft pipeline,)* result of former cycle initiated read -> GReg[1]

*(Soft pipeline,)* initiate DM[ G[0] ] read access. Data available next cycle.

GReg[1] -> ShLoop

GReg[0] - 1 -> GReg[0]

**Assembler**

Syntax: none as this instruction can not be used outside of the ROM context switch routine.

CPU Flags: T

Cycles: 1

Description: .

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |

Instruction Fields:

# ldShPC --Context Switch specific instruction--

-- THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE--

**Operation:**

*(Soft pipeline,* result of former cycle initiated read -> GReg[1]

*(Soft pipeline,* initiate DM[ G[0] ] read access. Data available next cycle.

GReg[1] -> ShPC

GReg[0] - 1 -> GReg[0]

**Assembler**

Syntax: none as this instruction can not be used outside of the ROM context switch routine.

CPU Flags: T

Cycles: 1

Description: .

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |

Instruction Fields:

# ldmGReg7 --Context Switch specific instruction--

-- THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE--

**Operation:**

*(Soft pipeline)* result of former cycle initiated read -> GReg[1]

*(Soft pipeline)* initiate DM[ G[0] ] read access. Data available next cycle.

GReg[1] -> GReg[7]

GReg[0] - 1 -> GReg[0]

**Assembler**

Syntax: none as this instruction can not be used outside of the ROM context switch routine.

CPU Flags: T

Cycles: 1

Description: .

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |

Instruction Fields:

# IdmGReg6--Context Switch specific instruction--

-- THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE--

**Operation:**

*(Soft pipeline)* result of former cycle initiated read -> GReg[1]

*(Soft pipeline)* initiate DM[ G[0] ] read access. Data available next cycle.

GReg[1] -> GReg[6]

GReg[0] - 1 -> GReg[0]

**Assembler**

Syntax: none as this instruction can not be used outside of the ROM context switch routine.

CPU Flags: T

Cycles: 1

Description: .

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |

Instruction Fields:

# ldmGReg5--Context Switch specific instruction--

-- THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE--

**Operation:**

*(Soft pipeline)* result of former cycle initiated read -> GReg[1]

*(Soft pipeline)* initiate DM[ G[0] ] read access. Data available next cycle.

GReg[1] -> GReg[5]

GReg[0] - 1 -> GReg[0]

**Assembler**

Syntax: none as this instruction can not be used outside of the ROM context switch routine.

CPU Flags: T

Cycles: 1

Description: .

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |

Instruction Fields:

# IdmGReg4--Context Switch specific instruction--

-- THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE--

**Operation:**

*(Soft pipeline)* result of former cycle initiated read -> GReg[1]

*(Soft pipeline)* initiate DM[ G[0] ] read access. Data available next cycle.

GReg[1] -> GReg[4]

GReg[0] - 1 -> GReg[0]

**Assembler**

Syntax: none as this instruction can not be used outside of the ROM context switch routine.

CPU Flags: T

Cycles: 1

Description: .

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |

Instruction Fields:

# ldmGReg3--Context Switch specific instruction--

-- THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE--

**Operation:**

*(Soft pipeline)* result of former cycle initiated read -> GReg[1]

*(Soft pipeline)* initiate DM[ G[0] ] read access. Data available next cycle.

GReg[1] -> GReg[3]

GReg[0] - 1 -> GReg[0]

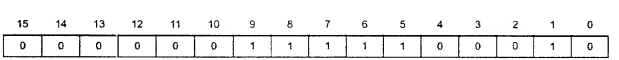
**Assembler**

Syntax: none as this instruction can not be used outside of the ROM context switch routine.

CPU Flags: T

Cycles: 1

Description: .



| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |

Instruction Fields:

# ldmGReg2--Context Switch specific instruction--

-- THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE--

**Operation:**

*(Soft pipeline)* result of former cycle initiated read -> GReg[0]

*(Soft pipeline)* initiate DM[ G[0] ] read access. Data available next cycle. GReg[1] -> GReg[2]

**Assembler**

Syntax: none as this instruction can not be used outside of the ROM context switch routine.

CPU Flags: T

Cycles: 1

Description: .

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |

Instruction Fields:

# ldmGReg1GReg0--Context Switch specific instruction--

-- THIS INSTRUCTION CAN NOT BE USED OUTSIDE OF THE ROM-CONTEXT SWITCH ROUTINE--

**Operation:**

*(Soft pipeline)* result of former cycle initiated read -> GReg[0]

GReg[0] -> GReg[1]

**Assembler**

Syntax: none as this instruction can not be used outside of the ROM context switch routine.

CPU Flags: none

Cycles: 1

Description: .

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |

Instruction Fields:

# cpShReg

**Assembler**
Syntax: `cpShReg.`

CPU Flags: none

Cycles: 1

Description: SF, RPC, T, PC registers are updated according to the value of their corresponding bits in the ShPC register.LM, EPC, DF, SPC registers are updated according to the value of their corresponding bits in the ShLoop register.

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |

Instruction Fields:

**DECLARATION AND POWER OF ATTORNEY**
**FOR PATENT APPLICATION**

Docket No. **CS 10246**

As a below named inventor, I hereby declare that:

My residence, post office address and citizenship are as stated below next to my name.

I believe I am the original, first and sole inventor (if only one name is listed below) or an original, first and joint inventor (if plural names are listed below) of the subject matter which is claimed and for which a patent is sought on the invention entitled **INTEGRATED PROCESSOR PLATFORM SUPPORTING WIRELESS HANDHELD MULTI-MEDIA DEVICES**, the specification of which is attached hereto unless the following box is checked:

☐ was filed on _____

as Application No. _____

and was amended on _____.

I hereby state that I have reviewed and understand the contents of the above identified specification, including the claims, as amended by any amendment referred to above.

I acknowledge the duty to disclose information that is material to the examination of this application in accordance with Title 37, Code of Federal Regulations, §1.56.

I hereby claim foreign priority benefits under Title 35, United States Code, § 119(a)-(d) of any foreign application(s) for patent or inventor's certificate listed below and have also identified below any foreign application for patent or inventor's certificate having a filing date before that of the application on which priority is claimed.

| Prior Foreign Application(s) | | | Priority Claimed |
|---|---|---|---|
| _____ | _____ | _____ | ☐ Yes ☐ No |
| (Serial No.) | (Country) | (Day/Month/Year Filed) | |
| _____ | _____ | _____ | ☐ Yes ☐ No |
| (Serial No.) | (Country) | (Day/Month/Year Filed) | |

I hereby claim the benefit under Title 35, United States Code, § 119(e) of any United States provisional application(s) listed below.

| | |
|---|---|
| _____ | _____ |
| (Serial No.) | (Filing Date) |

I hereby claim the benefit under Title 35, United States Code, § 120 of any United States application(s) listed below and, insofar as the subject matter of each of the claims of this application is not disclosed in the prior United States application in the manner provided by the first paragraph of Title 35, United States Code, § 112, I acknowledge the duty to disclose information which is material to patentability as defined in Title 37, Code of Federal Regulations, § 1.56 which became available between the filing date of the prior application and the national or PCT international filing date of this application.

| (Serial No.) | (Filing Date) | (Status - patented, pending, abandoned) |
|---|---|---|
| (Serial No.) | (Filing Date) | (Status - patented, pending, abandoned) |

I HEREBY APPOINT THE FOLLOWING AS MY ATTORNEY(S) OR AGENT(S) WITH FULL POWER OF SUBSTITUTION TO PROSECUTE THIS APPLICATION AND TRANSACT ALL BUSINESS IN THE PATENT AND TRADEMARK OFFICE CONNECTED THEREWITH:

| NAME(S) | REGISTRATION NO.(S) | ASSOCIATE POWER OF ATTORNEY ATTACHED |
|---|---|---|
| Michael C. Soldner | 41,455 | |
| Rolland R. Hackbart | 28,217 | |
| Sylvia Y. Chen | 39,633 | |

| Yes | No |
|---|---|
| | X |

Address all written correspondence to:

Motorola, Inc.
Personal Communications Sector
Intellectual Property Department (MCS)
600 North US Highway 45, Rm. AN475
Libertyville, IL 60048

Direct Telephone Calls to:

Michael C. Soldner
at (847) 523-2585

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

| FULL NAME OF FIRST INVENTOR | INVENTOR'S SIGNATURE | DATE |
|---|---|---|
| Sheila M. Rader | | |
| RESIDENCE | CITIZENSHIP | |
| 33248 N. Cove Rd., Wildwood, IL 60030 | U.S. | |
| POST OFFICE ADDRESS | | |
| same as above | | |

| FULL NAME OF SECOND INVENTOR | INVENTOR'S SIGNATURE | DATE |
|---|---|---|
| Brian Lucas | | |
| RESIDENCE | CITIZENSHIP | |
| IL | U.S. | |
| POST OFFICE ADDRESS | | |
| same as above | | |

| FULL NAME OF THIRD INVENTOR | INVENTOR'S SIGNATURE | DATE |
|---|---|---|
| Pradeep Garani | | |
| RESIDENCE | CITIZENSHIP | |
| Residence Agora, 2 Chem Henri Bosco, 31000 Toulouse, France | India | |
| POST OFFICE ADDRESS | | |
| same as above | | |

| FULL NAME OF THIRD INVENTOR | INVENTOR'S SIGNATURE | DATE |
|---|---|---|
| Franz Steininger | | |
| RESIDENCE | CITIZENSHIP | |
| 1077 E. Bart, Gilbert, AZ 85224 | Germany | |
| POST OFFICE ADDRESS | | |
| same as above | | |